ESD-TR-76-359

SECURITY KERNEL SPECIFICATION FOR A SECURE COMMUNICATIONS PROCESSOR

Honeywell Incorporated
13350 U.S. Highway 19
St. Petersburg, FL 33733

September 1976

Approved for Public Release;
Distribution Unlimited.

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
ELECTRONIC SYSTEMS DIVISION
HANSCOM AIR FORCE BASE, MA 01731

This technical report has been reviewed and is approved for publication.

William R. Price
WILLIAM R. PRICE, Capt, USAF
Techniques Engineering Division

Donald P. Eriksen
DONALD P. ERIKSEN
Techniques Engineering Division

FOR THE COMMANDER

Frank J. Emma
FRANK J. EMMA, Colonel, USAF
Director, Computer Systems Engineering
Deputy for Command & Management Systems

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ESD-TR-76-359	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SECURITY KERNEL SPECIFICATION FOR A SECURE COMMUNICATIONS PROCESSOR		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) C. H. Bonneau		8. CONTRACT OR GRANT NUMBER(s) F19628-74-C-0193
9. PERFORMING ORGANIZATION NAME AND ADDRESS Honeywell Incorporated 13350 U.S. Highway 19 St. Petersburg, Florida 33733		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Deputy for Command and Management Systems Electronic Systems Division, Hanscom AFB, MA 01731		12. REPORT DATE September 1976
		13. NUMBER OF PAGES 145
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; Distribution Unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer Security
Secure Computer Systems
Security Kernel
Secure Communications Processor
Secure Front-End Processor
HIS Level 6

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report presents a formal top level interface specification of a kernel for a secure communications processor based on a Honeywell Level 6/40 minicomputer enhanced with a security protection module (SPM).

PREFACE

Because of funding limitations, the Air Force terminated the effort which this document describes before the effort reached its logical conclusion. This report is incomplete, but was published in the interest of capturing and disseminating the computer security technology that was available when the effort was terminated.

This report describes the top level design of a security kernel for a secure communications processor. Although the kernel described in this report should be useful in a wide range of minicomputer applications, the kernel was initially intended for use in the front-end processor of a secure, general purpose computer system (Multics). In general, the kernel functions described in this report appear adequate. The Air Force review of this document included comments on particular design issues and the style of specification and the resulting impact of this style on verification of the design. Because sufficient funds to completely revise the report were not available, the outstanding comments appear in an appendix.

ACKNOWLEDGMENTS

The author has drawn extensively from the ideas and suggestions presented in the Multics Security Kernel Top Level Specification in the preparation of this report and gratefully acknowledges the contribution of its author, Jerry Stern. The author would also like to give special thanks to Lee Scheffler for many helpful discussions over the past year.

TABLE OF CONTENTS

1.0 INTRODUCTION

This report presents a formal top level interface specification of a kernel for a secure communications processor based on a Honeywell Level 6/40 minicomputer enhanced with a security protection module (Reference 3). This kernel provides access controls to support the security and integrity access policies as defined in the Bell and LaPadula model (Reference 1).

This specification is intended to be suitable for formal proof of correctness. The specification is written in the formal specification and assertion language, SPECIAL, developed by Stanford Research Institute (Appendix B).

Implementation of this specification is intended to support both secure front-end processor applications and general-purpose stand-alone communications applications.

2.0 SPECIAL OVERVIEW

The following description is intended to serve only as an overview of some of the basic features of SPECIAL. For a complete description of the language, the reader is referred to the SPECIAL reference manual written by Stanford Research Institute and included as Appendix B of this document.

SPECIAL allows for the definition of abstract data objects and operations performed upon these objects. The objects are represented as V-functions, i.e., functions that return a value. The collection of V-functions represents the state of the system being specified. Operations on objects are represented by O-functions. O-functions modify the values of V-functions and thereby change the state of the system. A third class of functions is that of OV-functions, i.e., functions that both change the system state and return a value.

A distinction among V-functions is that they can be either primitive or derived. The value of a derived V-function is simply an expression in terms of other V-functions. Only the values of primitive V-functions can be changed by O-functions and this implicitly changes the values of related derived V-functions.

A second distinction among V-functions is that they can be either visible or hidden. A visible V-function is one that is available to the outside world. A hidden V-function can only be referenced from within other functions. By convention, the names of all hidden V-functions contained in the kernel specification begin with the prefix "h_".

An individual function specification comprises several different parts. For visible functions, the first part is always a list of exception conditions, i.e., conditions under which the function fails to operate or return a value. These exceptions apply only when a visible function is referenced from the outside world. When referenced internally, i.e., from within another function, exceptions are ignored. Similarly, hidden functions, which can only be referenced internally, have no exceptions. The order of exceptions is significant. Each exception condition is checked in turn, and only if all the conditions are false will the effects take place and/or the value be returned. The calling program may depend on the order in which the exception conditions are checked. The remaining parts of a function specification differ for V-functions and O-functions. In the case of a V-function, the specification contains either an initial value for a primitive V-function or the derivation expression for a derived V-function. O-functions and OV-functions contain a list of "effects" that describe changes to V-functions. For an OV-function, the effects also define the output value. The order of effects within a given list is unimportant. All effects occur at once, i.e., instantaneously. Within an effects list, references are made to the old and new values of V-functions, i.e., the values before and after the effects have occurred. The new value of a V-function is indicated by a single quotation mark (') preceding the V-function name.

The entire set of functions which constitute the top level specification are divided into groups called modules. The purpose of modules is to allow the specifier to conveniently organize a possibly large number of functions into small groupings that can be easily understood. The modularization serves no other purpose. Often, the modules may, in some ways, correspond to actual program modules in a perceived implementation. This, however, is not necessary.

Function references between modules are permitted. A function of one module can observe the value of V-functions, including hidden V-functions, of any other module. However, V-functions can only be directly modified by O- or OV-functions of the same module. Therefore, in order for one module to change a V-function of another module, the first module must invoke an O- or OV-function of the second module.

The specification of a module is divided into six sections as follows:

1. TYPES

   This section defines new data types which supplement the primitive data types of the language.

2. DECLARATIONS

   This section defines variable names and their associated data types.

3. PARAMETERS

   This section defines named constants which, in SPECIAL, are called parameters.

4. DEFINITIONS

   This section allows for the specification of macros.

5. EXTERNALREFS

   This section defines all external functions referenced within the module.

6. FUNCTIONS

   This section contains all of the individual function specifications for the module.

3.0 SECURITY KERNEL SPECIFICATION OVERVIEW

The SCOMP kernel specification is divided into the following modules:

- clock
- access_levels
- processes
- volumes
- quota_cells
- segments
- devices
- address_spaces
- host_interfaces

Each module takes the form of an object manager; i.e., each module defines the data representation and operations for a particular object type.

The functions of these modules define the top level kernel interface. The visible kernel interface includes all non-hidden V-functions and all O- and OV-functions not specifically excluded from the interface by the SCOMP-kernel interface specification which precedes the module specifications.

The SCOMP-kernel interface specification identifies the modules that compose the interface and, for each module, identifies the O- and OV-functions that are excluded from the interface. Unlike the module specifications, the interface specification is not written in SPECIAL, but is written in a specific interface specification language also developed by Stanford Research Institute. Functions excluded from the interface are specified following the WITHOUT clause for each module.

4.0 SCOMP KERNEL MODULES

4.1 Clock

The first module of the SCOMP kernel specification describes the system real-time clock. The read_real_clock function returns the current value of the system real-time clock. The clock is used as a source of unique identifiers by the various kernel modules.

4.2 Access Levels

The access_levels module serves to define the internal structure of an access level. An access level is composed of both a security level and an integrity level. A security level or an integrity level is composed of both a level number (e.g., confidential, secret) and a category set (e.g., NATO, crypto).

The access_levels module contains functions for determining subject access (read, write, or read/write) to an object. These functions implement the nondiscretionary access control policies of the security model (simple security and the *-property). The functions take as input a subject access level, an object access level, and a boolean value indicating whether the subject is trusted. All access control decisions made by the kernel depend on these functions.

4.3 Processes

The processes module is concerned with the creation and deletion of processes, interprocess communication, process scheduling priority, process virtual clocks, and process real timers.

At process creation, the access level of the new process, the trusted or untrusted status of the new process, and the scheduling priority of the new process are specified. Process creation and deletion are restricted to the trusted initializer process only. For this reason, process creation cannot serve as an information path between untrusted processes. Therefore, knowledge of the existence of processes is not restricted.

The processes module provides an interprocess communication mechanism via the block, read-messages, send-signal, and wake-up functions. The send-signal and wake-up functions insert a message into the target process message queue. If the target process state is blocked, the process state is changed to ready. The difference between signals and wake-ups is that a signal is recognized immediately by the receiving process whereas a wake-up is not recognized until the receiving process next invokes the block or read_messages functions. The block function allows a process to read its message queue; if the queue is empty, the process is blocked (state changes to block) until a message arrives. A read_messages function is provided that simply reads any pending messages, but will not wait for a message to arrive.

The processes module supports the process scheduling mechanism, which is hidden within the kernel, by maintaining the scheduling priority of each process. Process priority is modified via the set_priority function. A process may change its own priority without any restrictions being imposed by the kernel; however, the priority of another process may only be increased. The purpose of this mechanism is to allow a process to associate a message with a priority. A message may be sent to another process via the wake-up or send_signal functions. The sender may then request that the scheduling priority of the target be changed to that of the message. If the message priority is greater than the current target process scheduling priority, its priority is raised to that of the message.

The process module supports real-time timers; the kernel maintains one real timer per process. If additional timers are required by a process, they must be managed by the clock manager portion of the supervisor. When a timer is set, the caller specifies the timer-interval, a wake-up message (timer identifier), and the interrupt level at which the clock manager is to run. When a timer runout occurs, the target process is sent a wake-up message and the corresponding timer level indicator is set. Upon process dispatch, the process will execute the level handler corresponding to the highest priority level indicator which is set.

The processes module also supports a per process virtual-time clock which only runs when the process state is running. Ideally, process virtual time represents the real time that a process has actually been running on a processor with subtractions made for certain hidden events such as page faults and interrupts. The objective is to factor out time spent performing hidden operations that support the virtual process environment because this time is unpredictable and, in general, depends upon the activities of other processes. The implementation of this ideal virtual clock is difficult. Thus, the kernel allows access to the virtual clocks only to the trusted time-monitor process for accounting purposes.

4.4 Volumes

The volumes module is concerned with the creation and deletion of volumes, and the mounting and demounting of volumes. Volumes are considered a logical subdivision of secondary storage consisting of a single disc pack.

At volume creation, minimum and maximum access levels for the volume are specified. These access levels define the range of access levels of information (i.e., segments) that can be stored on the volume. The minimum access level of the volume also serves as a visibility access level for the volume. Volume creation and deletion are restricted to the trusted volume initializer process to allow a limit to be placed on the number of volumes without creating an information channel. At volume creation, an initial quota cell and quota cell limit are created for the volume. The quota value of the initial quota cell is set to the volume size (i.e., the number of pages on the volume). The quota limit value of the initial quota cell limit is set to the number of allowed quota cells per volume. Details on quota cells may be found in the description of the quota_cell module.

The only operations which can be performed on a volume by untrusted processes are mounting and demounting. Both of these operations require that the caller's access level be equal to the volume minimum access level because the mounting and demounting of a volume can be detected by any process at an access level equal to or greater than the volume minimum access level. To prevent the creation of an information channel caused by the finite number of disc drives in the system, disc drives are allocated by access level. The kernel requires that for a process to mount a volume on a disc drive, the process access level, the volume minimum access level, and the drive access level must all be equal. The access level distribution of free disc drives may be changed dynamically, if necessary, by the trusted initializer process using the change_drive_count function.

4.5 Quota Cells

The quota_cells module is concerned with the management of volume quota and provides the mechanism to build a hierarchical file system outside the kernel. Quota cells are the mechanism by which the sharing of pages on a volume is controlled - each segment is assigned to a particular quota cell. The kernel provides a general interface that allows any arbitrary collection of segments having the same access level to be assigned to the same quota cell. The supervisor (i.e., file manager) is responsible for imposing an arrangement of quota cells and segments that reflects a directory hierarchy.

At quota cell creation, a volume, an access level, and a visibility access level are specified. The two access levels must be within the volume range. Storage space on a volume is first made available by the creation of an initial quota cell at volume creation. Both the access level and the visibility access level of the initial quota cell must equal the volume minimum access level permitting the initial quota to be distributed to higher access levels as needed. All other quota cells are created with a quota of zero pages. In order to assign a nonzero quota to a new quota cell, quota must be moved from some other quota cell on the same volume. Quota cannot be moved downward by untrusted processes - the move_quota function requires that the target quota cell have an equal or greater access level than the source quota cell; only trusted processes may move quota downward.

A reference count is associated with each quota cell. This count is the number of segments that charge pages to the quota cell. As long as this count is nonzero, the quota cell cannot be deleted. Another attribute of a quota cell is the count of pages used. This count is changed by the creation and deletion of segments (segment length is converted to equivalent pages) as specified in the segments module. For all quota cells, the pages used count is not permitted to exceed the quota.

To prevent the creation of an information path resulting from shared finite resources, a limit is placed on the number of quota cells that can be created on a volume. At volume creation, the volume quota cell limit is assigned to the volume minimum access level permitting the volume limit to be distributed to higher access levels as needed. All other access levels quota limits are initially zero. In order to create a new quota cell with a visibility access level greater than the volume minimum access level, the quota cell limit must be redistributed from some lower access level limit on the volume. The quota cell limit cannot be moved downward by untrusted processes - the change_quota_cell_limit function requires that the target quota cell limit have a greater visibility access level than the source quota cell limit; only trusted processes may change quota cell limits downward.

4.6 Segments

The segments modules is concerned with the management of segments, the logical unit of storage for SCOMP. Within this module, segments are identified by global identifier only; the local process view of segments are identified by segment number, as defined in the address_spaces module. The segments module also specifies the interface for the SCOMP memory manager process which is responsible for all memory manager policy decisions.

At segment creation, a visibility access level, a volume, a quota cell, and a segment length must be specified. The new segment resides on the specified volume and charges its pages to the specified quota cell. The access level of the new segment is equal to that of its associated quota cell. The number of pages charged to its quota cell is derived directly from the requested segment length; initially all words of a segment have a value of zero. Only paged segments may be dynamically created by a user; no main memory allocation is made at segment creation. No limit is placed on the number of segments that can be created on a given volume.

For flexibility, a memory manager interface has been included in the kernel specification in the segments module. The memory manager is to be an untrusted process running at system high which is responsible for all memory management policy decisions; the kernel provides the mechanisms for implementing its policy decisions. Thus, policy may change without affecting the kernel.

The memory manager interface must be flexible in order to accommodate the various operating environments for SCOMP. That is, the kernel specification must be applicable to the following configurations:

- High performance, core-only system with sufficient memory to preallocate memory at load time; for performance reasons, unpaged segments are utilized as much as possible.
- High performance, core-only system with insufficient memory to preallocate all data buffers at load time; thus, a simple buffer manager is required.
- Medium performance, disc-supported, fully-paged virtual memory system with a general memory management function.

The kernel specification satisfies the requirements imposed by the various configurations, yet retains a simple user interface, by imposing the following rules and conventions:

- Paging is invisible to all user processes; visibility of paging is restricted to the memory manager. The only reason for the visibility of paging effects within the specification is to support the memory manager interface.
- Unpaged segments must be defined at load time and retained in core until deleted; only paged segments are subject to dynamic memory management.
- The allocation of real memory to requestors is postponed as long as possible by the kernel; no real memory is allocated until the user attempts to access it.

The kernel design envisioned is as follows:

- The kernel has a free list of available memory pages from which it draws to satisfy access requests (allocate_page function).
- When the number of free pages falls below some fixed threshold (min_free_page_count), the external memory manager is requested to provide additional pages (page_ request_count).
- When the kernel is unable to satisfy an access request, the requestor is delayed until free resources are available (as supplied by the memory manager).

The kernel specification provides a memory manager interface via two functions: get_memory_data and provide_free_pages. These functions are only available to the memory manager.

The get_memory_data function provides a list of pages eligible for memory deallocation. The memory utilization data consists of the global segment name, the page number, the page core address, an indication of whether the page has been used since the last time the function was called, and an indication of whether the page has been modified since it was placed in main memory. To be eligible for deallocation, a page must be in core, and not wired. The provide_free_pages function notifies the kernel of the pages selected by the memory manager for deallocation. These pages are swapped to disc if modified and placed on the free list. The data provided by the kernel to the memory manager process appears sufficient to support a reasonable memory management policy, e.g., frequency of page use is provided along with method of access (read or write) - thus, a reasonable policy would be to favor pages used infrequently in a read mode only.

Functions are defined for reading and writing the contents of segments (i.e., read/read). These functions have certain side effects, as well. That is, if the target segment is paged, then the kernel checks the page_in_core indicator. If not in core, a free page is allocated to the process to satisfy the request. If the free page count falls below some minimum threshold, the memory manager is sent a wake-up.

Segment deletion requests are delayed until all outstanding I/O on the segment has completed. Upon satisfaction of this condition, and if no exception conditions are true, pages in core (paged segment only) are added to the free list. Similarly, an unpaged segment (which must be in core) is converted to equivalent pages and added to the free list. The page quota of the quota cell associated with the segment is increased by the equivalent segment length.

To support memory management, the segments module is responsible for monitoring pages used, pages modified, and segments wired on a global basis. Since page descriptors are to be shared, page used and modified indicators are only maintained within the segments module, no process local view of page utilization (within the address_spaces module) is necessary. However, wired segments are locally known (a segment must be locally wired to perform I/O). The segments module's global view of wired segments is derived from the local data and provides a count of the number of processes which have the segment wired. The hidden change_wire_count function is called from address_spaces whenever a segment is locally wired (wire_seg) or unwired (unwire_seg).

4.7 Devices

The devices module is responsible for the management of devices. Within this module, devices are identified by global identifier only; the local process view of devices, where devices are identified by device number, is defined in the address_space module. The device module also specifies the interface for the SCOMP device monitor process.

At device creation, the device access level, the device type, and the device mapping mode are specified. Since all devices are read-write from a security viewpoint (all operations may return status information visible to a using process and may result in a change of state visible to the external world), a process must have an access level equal to that of the device in order to initiate it. The device type is used to define those read and write control operations valid for user processes to perform with the device. The mapping mode specifies whether the device is to operate mapped or premapped. Device creation and deletion are restricted to the trusted device manager process. Similarly, the modification of device attributes (access level and mapping mode) is restricted to the device manager.

For completeness, a device monitor interface has been included in the kernel specification in the devices module. The device monitor process would be analogous to the memory manager process. The get_device_data function provides the necessary interface. The function provides the device utilization data to the device monitor. This information consists of the device global name, a device initiated indicator, an indication of whether the device has been used since the function was last called, and an indicator of whether the device has been written to since the function was last called for all devices which exists in the global device catalog.

The devices module also includes the device_wake-up function which notifies the device initiator process of asynchronous I/O termination. When the termination interrupt is received, the kernel sends the initiator process a wake-up message consisting of the device local name and sets the corresponding device level indicator which schedules the device handler for the process.

4.8 Address Spaces

The address_spaces module supports both the binding of segment numbers to segments and device numbers to devices within a process as well as the granting and revoking of current access to segments and devices. In addition, this module provides kernel interface functions for reading and writing segments and devices, as well as functions to support trap and level handling within a process.

The binding of a segment number to a segment is accomplished via the assign_segno function. The assigned segment number is selected by the supervisor, not the kernel. Information about assigned segments is maintained in a per-process table called the known segment table (kst). The kst is represented by a collection of V-functions. Similarly, the binding of a device number to a device is accomplished via the assign_devno function. The assigned device number is selected by the supervisor, not the kernel. Information about assigned devices is maintained in a per-process table called the known device table (kdt). The kdt is represented by a collection of V-functions.

The assignment of a segment number to a segment or a device number to a device does not make the segment or device directly accessible. Object access is accomplished via the give_access and give_device_access functions for which the caller specifies the access mode and ring brackets for the segment or device. The requested access mode is modified, if necessary, by the kernel to conform to access level restrictions. For device access, a process level must also be defined (at which a device handler must have been previously defined) in order to allow the process execution point to be properly handled during asynchronous I/O termination notification (i.e., device_ wake-up). Any process can grant itself access to a segment or device in this fashion. However, it is expected that the supervisor will control the use of these functions so as to enforce the discretionary access control list (ACL) policy. Similarly, the supervisor will enforce the discretionary ring bracket policy. The kernel requires only that the segment and device ring brackets be outside the kernel ring.

Functions are also provided to revoke the access of all processes to a given segment or device. These functions can be used by the supervisor to force all processes to recompute access to a segment or device after an ACL has been changed. Once access has been revoked, subsequent references to the segment or device will cause an exception. The address_spaces module also includes a revoke_vol_access function. At the time a volume is demounted, access to all segments on the volume is revoked by the use of this function.

The address_spaces module provides two functions, hidden from the kernel interface, which support process creation and deletion. The init_address_space function initializes the new process address space at process creation time. The process kst is initialized to a template_kst and the initial process execution point is defined. This process initialization is identical for all processes. It is envisioned that the new process will then utilize the interprocess communications mechanism to communicate with the initializer process (e.g., answering service process) to obtain sufficient data to establish its process-unique functionality (e.g., create its kdt). The purge_address_space function releases all segments and devices in the address space of the process at process deletion time.

Several functions in the address_spaces module represent either memory reference or device reference operations associated with classes of machine instructions. These functions include:

- Read_seg - reads a word from a segment for process
  - represents all load memory instructions
- Write_seg - writes a word to a segment for process
  - represents all store memory instructions
- Execute_seg - executes a word from a segment for process
  - represents all fetch instruction from memory cycles
- Call - initiates inter-ring inward movement for process
  - represents Link Jump across Rings (LNJR) machine instruction

- Return - initiates inter-ring outward movement for process
  - represents Return (RETN) machine instructions
- Sync_device_read - reads a word from a device synchronously for process
  - represents all PIO input instructions (IO, IOH)
- Sync_device_write - writes a word to device synchronously for process
  - represents all PIO output instructions (IO, IOH)
- Connect_device_read - starts an asynchronous read operation from device for process
  - represents all DMA input connect instructions (IOLD)
- Async_device_read - asynchronously reads a word from mapped device to segment for process
  - represents DMA transfer from device to memory (IOLD execution)
- Connect_device_write - starts an asynchronous write operation to device for process
  - represents all DMA output connect instructions (IOLD)
- Async_device_write - asynchronously writes a word from segment to mapped device for process
  - represents DMA transfer from memory to device (IOLD execution)

The address_spaces module also includes two functions primarily intended to support secure user-intiated I/O. A segment must be maintained in main memory (no page faults) during DMA I/O to prevent security violations. The function wire_seg provides the mechanism by which a user may insure that an entire segment is in main memory and no longer eligible for deallocation by the memory manager. The function unwire_seg releases a wired segment. The effects of wire_seg are to set a process-local wire indicator (h_kst_seg_wired) on the segment descriptor and a global wire indicator (h_seg_wire_count). The global indicator is used to support the memory manager interface while the local indicator is checked by the hardware before allowing a connect_device operation to proceed. In order to have a process-local mechanism to mediate unwire_seg requests, yet support multiple devices performing I/O on the same segment, an I/O count (h_kst_seg_io_count) is also maintained on the process-local segment descriptor. The I/O count is incremented (by hardware) whenever a DMA transfer is initiated and decremented (by hardware) upon I/O termination (i.e., effects of device_wake-up).

The address_spaces module allows a process to execute at multiple interrupt priority levels within its address space. Three visible functions are adequate to support this capability. The set level handler function allows a user to specify a level handler (i.e., task handler) for a process at a given interrupt priority level. The kernel enforces the discretionary ring bracket policy within this function as follows: (a) the effective ring of execution of the level handler cannot be more privileged than the caller ring; and, (b) if a more privileged level handler already exists, then the request is denied (if the existing handler is not more privileged, it is replaced). The modify_activity_level and the lev functions allow a user to manipulate the activity level indicators for his process. They represent the functionality provided by the level change (LEV) machine instruction. The modify_activity level function reflects the defer interrupt option of the LEV instruction. The lev function reflects both the normal LEV instruction (level context switch) execution and its suspend level option. The only requirement for these functions is that the level to be manipulated must have a defined handler. To support the multiple priority level capability, three functions hidden from the kernel interface are included. These functions are: set_level, change_level, and dispatch_level.

The last feature of the address_spaces module is trap handling or fault processing. The visible trap handling functions allow nonkernel software to handle traps; certain traps (e.g., page faults) are not visible at the kernel interface and are not covered by these visible functions. The trap structure supported by the SCOMP kernel is a generalization and simplification of the particular trap mechanism implemented in the SCOMP hardware. The set_trap_handler function allows the user to specify a trap handler and trap save area for a particular trap type for his process. The kernel enforces the discretionary ring bracket policy within this function in a manner similar to level handlers. That is, (a) the effective ring of execution of the trap handler cannot be more privileged than the caller ring; and, (b) if a more privileged trap handler already exists, then the request is denied (if the existing handler is not more privileged, it is replaced). The trap function specifies the behavior of the kernel upon the occurrence of a given trap type. The kernel passes the pertinent trap information into the user defined trap save area and changes the process execution point to the trap handler. The return_from_trap function specifies the return from the user-defined trap handler upon completion of fault processing. The only significant restriction imposed by the kernel within this function is that the ring of execution of the specified return execution point not be more privileged than that of the trap handler; it is not necessary to return to the point where the trap occurred.

4.9 Host Interfaces

The host_interfaces module provides those functions necessary to support secure communication between a host (e.g., Multics) and a front-end processor (SFEP). As specified, the link between the SFEP process and its associated host process is the device name (e.g., terminal) on whose behalf both processes were created. That is, neither the front-end process nor the host process need be aware of the others process identifier in order to communicate since only one untrusted user process may have a device initiated at any one time. Thus, specifying the device name implies a unique target process.

In order to support the host interface, several kernel data areas are associated with each device, represented by a collection of V-functions. These are:

- device_status_area - Area used by the SFEP process to transmit device status information to the host process.
- device_control_area - Area used by the host process to provide device control information to the SFEP process (e.g., change translation table).
- message_queue - Area used by the SFEP process to send messages to its associated host process.

In addition, each device has a data area or buffer in user space whose location is specified to the kernel when data is to be sent to the host process or when data is to be received from the host.

The following SFEP kernel functions are included in the host_ interfaces module to support the host interface:

- send_message - Allows the user to transfer a data buffer (i.e., message) to its associated host process.
- receive_message - Allows a user to specify a data buffer to receive a data message from its associated host process.

- read_control_data - Allows a user to read the device control table set by the host process.
- write_status_data - Allows a user to transfer device status information to its associated host process.
- send_wake-up - Allows a user to send a message to its associated host process via its message queue. This function is analogous to the wake-up function which supports SFEP inter-process communication.
- send_host_signal - Allows a user to interrupt the host process to receive immediate attention. This function is intended to support a device "QUIT" mechanism.

5.0 REFERENCES

1. Computer Security Model: Unified Exposition and Multics Interpretation, D. E. Bell/L. J. LaPadula, MTR-2997, July 1975, MITRE Corporation, Bedford, Mass.

2. Secure Communications Processor Specification, ESD-TR-76-351, Vol II, R. Broadbridge/J. Mekota, June 1976.

3. Detail Specification for the Security Protection Module (SPM), ESD-TR-76-366, G. Rolfe/J. Carnall, September 1976.

Appendix A

SCOMP Kernel Specification

DS 34028917

14 February 1977

Rev. B 25 May 1977

Honeywell

AVIONICS DIVISION
ST. PETERSBURG, FLORIDA

```
(
INTERFACE  SCOMP_kernel

(clock)

(access_levels)

(processes          WITHOUT   dispatch
                              wake
                              advance_virtual_clock)

(volumes)

(quota_cells        WITHOUT   set_quota
                              change_qc_refs
                              change_qc_pages_used
                              set_quota_cell_limit)

(segments           WITHOUT   read
                              write
                              allocate_page
                              deallocate_page
                              add_page_to_free_list
                              remove_page_from_free_list
                              change_wire_count
                              set_page_used_indicator
                              set_page_indicators)

(devices            WITHOUT   set_device_active
                              write_device
                              assign_device
                              release_device)

(address_spaces     WITHOUT   init_address_space
                              purge_address_space
                              revoke_vol_access
                              reset_device_used_indicator
                              reset_device_modified_indicator
                              decrement_seg_io_count
                              set_level
                              dispatch_level
                              change_level)

(host_interfaces)
)
```

```
MODULE  clock

     DECLARATIONS

INTEGER time;

     FUNCTIONS

VFUN read_real_clock() -> time;
     $(returns the value of the system-wide real time clock)
     INITIALLY time = 0;

OFUN advance_real_clock();
     $(advances real time clock one time unit)
     EFFECTS
       'read_real_clock() = read_real_clock() + 1;

OVFUN get_uid() -> time;
     $(generates a unique identifier)
     DELAY UNTIL read_real_clock() > h_last_uid();
     EFFECTS
       time = read_real_clock();
       'h_last_uid() = time;

VFUN h_last_uid() -> time;
     $(returns last unique identifier generated)
     HIDDEN;
     INITIALLY time = 0;

END_MODULE
```

```
MODULE  access_levels


     TYPES

level_number : {INTEGER ln : 0 <= ln AND ln <= max_ln};
category_set : {VECTOR_OF BOOLEAN cs : LENGTH(cs) = cs_size};
security_level : STRUCT (level_number sln;
                         category_set scs);
integrity_level : STRUCT (level_number iln;
                          category_set ics);
access_level : STRUCT (security_level sl;
                       integrity_level il);


     DECLARATIONS

access_level sub_al, ob_al;
INTEGER i;
BOOLEAN b, trusted;


     PARAMETERS

level_number max_ln $(maximum level number);
INTEGER cs_size $(category set size);


     FUNCTIONS

VFUN h_read_allowed(trusted; sub_al; ob_al) -> b;
     $(returns true if subject can read object)
     HIDDEN;
     DERIVATION sub_al.sl.sln >= ob_al.sl.sln AND
                (sub_al.il.iln <= ob_al.il.iln OR trusted) AND
                (FORALL i : i >= 1 AND i <= cs_size :
                (ob_al.sl.scs[i] => sub_al.sl.scs[i]) AND
                ((sub_al.il.ics[i] => ob_al.il.ics[i]) OR trusted));

VFUN h_write_allowed(trusted; sub_al; ob_al) -> b;
     $(returns true if subject can write object)
     HIDDEN;
     DERIVATION (sub_al.sl.sln <= ob_al.sl.sln OR trusted) AND
                sub_al.il.iln >= ob_al.il.iln AND
                (FORALL i : i >= 1 AND i <= cs_size :
                ((sub_al.sl.scs[i] => ob_al.sl.scs[i]) OR trusted) AND
                (ob_al.il.ics[i] => sub_al.il.ics[i]));

VFUN h_read_write_allowed(trusted; sub_al; ob_al) -> b;
     $(returns true if subject can read and write object)
     HIDDEN;
```

```
    DERIVATION h_read_allowed(trusted; sub_al; ob_al) AND
               h_write_allowed(trusted; sub_al; ob_al);

END_MODULE
```

```
MODULE  processes

     TYPES

level_number : {INTEGER ln : 0 <= ln AND ln <= max_ln};
category_set : {VECTOR_OF BOOLEAN cs : LENGTH(cs) = cs_size};
security_level : STRUCT (level_number sln;
                         category_set scs);
integrity_level : STRUCT (level_number iln;
                          category_set ics);
access_level : STRUCT (security_level sl;
                       integrity_level il);
process_uid : INTEGER;
message : INTEGER;
message_queue : VECTOR_OF message;
process_state : {running, ready, blocked};
process_priority : {INTEGER pr : 0 <= pr AND pr <= max_priority};
level : {INTEGER level : 1 <= level AND level <= pl_size};


     DECLARATIONS

process_uid procuid, newproc, target;
access_level al;
message msg;
message_queue msg_queue;
process_priority pr;
level level;
process_state state;
BOOLEAN b;
INTEGER i, n, time, delta_t, real_timer_id;


     PARAMETERS

process_uid initializer_id $(initializer process id);
process_uid time_monitor $(time monitor process id);
level signal_level $(process signal level);
INTEGER max_processes $(maximum number of processes);
INTEGER max_messages $(maximum size of process message queue);
INTEGER max_priority $(maximum process priority);


     DEFINITIONS

BOOLEAN no_process(procuid) IS
        ~h_proc_exists(procuid);

BOOLEAN write_not_allowed(procuid; al) IS
        ~h_write_allowed(h_proc_trusted(procuid), h_proc_al(procuid), al);
```

```
BOOLEAN no_message(procuid) IS
       LENGTH(h_proc_msg_queue(procuid)) = 0;

BOOLEAN invalid_timer(n) IS
       n <= 0;

BOOLEAN undefined_level(procuid; level) IS
       ~h_level_handler_exists(procuid, level);

BOOLEAN no_real_timer(procuid) IS
       ~h_proc_real_timer(procuid);


     EXTERNALREFS

FROM clock :
     VFUN read_real_clock() -> time;
     OVFUN get_uid() -> procuid;

FROM access_levels :
     level_number max_ln $(maximum level number);
     INTEGER cs_size $(category set size);
     VFUN h_write_allowed(b; al; al) -> b;

FROM address_spaces :
     level pl_size $(number of visible activity levels);
     VFUN h_level_handler_exists(procuid; level) -> b;
     OFUN init_address_space(procuid);
     OFUN purge_address_space(procuid);
     OFUN dispatch_level(procuid);
     OFUN set_level(procuid; level; b);


     FUNCTIONS

OVFUN create_proc(al; b; pr)[procuid] -> newproc;
     $(creates a new process)
     EXCEPTIONS
       procuid ~= initializer_id;
       h_proc_count() = max_processes;
     EFFECTS
       newproc = EFFECTS_OF get_uid();
       'h_proc_exists(newproc) = TRUE;
       'h_proc_al(newproc) = al;
       'h_proc_trusted(newproc) = b;
       'h_proc_priority(newproc) = pr;
       EFFECTS_OF init_address_space(newproc);
       'h_proc_count() = h_proc_count() + 1;

OFUN delete_proc(target)[procuid];
```

```
      $(deletes a process)
      EXCEPTIONS
        procuid ~= initializer_id;
        no_process(target);
      EFFECTS
        'h_proc_count() = h_proc_count() - 1;
        'h_proc_exists(target) = FALSE;
        EFFECTS_OF purge_address_space(target);

VFUN h_proc_exists(procuid) -> b;
      $(returns true if process exists)
      HIDDEN;
      INITIALLY b = FALSE;

VFUN h_proc_count() -> n;
      $(returns number of existing processes)
      HIDDEN;
      INITIALLY n = 0;

VFUN h_proc_al(procuid) -> al;
      $(returns access level of process)
      HIDDEN;
      INITIALLY al = ?;

VFUN proc_al()[procuid] -> al;
      $(external form of h_proc_al)
      DERIVATION h_proc_al(procuid);

VFUN h_proc_trusted(procuid) -> b;
      $(returns true if process is trusted)
      HIDDEN;
      INITIALLY b = ?;

VFUN proc_trusted()[procuid] -> b;
      $(external form of h_proc_trusted)
      DERIVATION h_proc_trusted(procuid);

VFUN h_proc_priority(procuid) -> pr;
      $(returns process priority)
      HIDDEN;
      INITIALLY pr = ?;

VFUN proc_priority()[procuid] -> pr;
      $(external form of h_proc_priority)
      DERIVATION h_proc_priority(procuid);

OFUN set_priority(target; pr)[procuid];
      $(sets process scheduling priority)
      EXCEPTIONS
        no_process(target);
        write_not_allowed(procuid, h_proc_al(target));
```

```
    EFFECTS
      'h_proc_priority(target)  = IF target = procuid
                                  THEN pr
                                  ELSE IF h_proc_priority(target) < pr
                                      THEN pr
                                      ELSE h_proc_priority(target);

OFUN dispatch(target);
    $(dispatches a process)
    EFFECTS
      'h_proc_state(target) = running;
      EFFECTS_OF dispatch_level(target);

VFUN h_proc_state(procuid) -> state;
    $(returns process state)
    HIDDEN;
    INITIALLY state = ?;

OFUN wake(target; msg);
    $(sends a message to target process
      message is lost if target queue full)
    DEFINITIONS
      INTEGER n IS LENGTH(h_proc_msg_queue(target));
      INTEGER m IS IF n = max_messages THEN n ELSE n+1;
    EFFECTS
      'h_proc_msg_queue(target) = VECTOR (FOR i FROM 1 TO m :
        IF i <= n THEN h_proc_msg_queue(target)[i] ELSE msg);
      'h_proc_state(target) = IF h_proc_state(target) = running
                              THEN running ELSE ready;

OFUN wakeup(target; msg)[procuid];
    $(external form of wake)
    EXCEPTIONS
      no_process(target);
      write_not_allowed(procuid, h_proc_al(target));
    EFFECTS
      EFFECTS_OF wake(target, msg);

OFUN send_signal(target; msg)[procuid];
    $(interrupt form of wakeup)
    EXCEPTIONS
      no_process(target);
      write_not_allowed(procuid, h_proc_al(target));
    EFFECTS
      EFFECTS_OF set_level(target, signal_level, TRUE);
      EFFECTS_OF wake(target, msg);

OVFUN read_messages()[procuid] -> msg_queue;
    $(returns contents of process message queue)
    EXCEPTIONS
      no_message(procuid);
      EFFECTS
        msg_queue = h_proc_msg_queue(procuid);
        'h_proc_msg_queue(procuid) = VECTOR ();

  OVFUN block()[procuid] -> msg_queue;
      $(returns contents of process message queue
        if queue empty, waits for message to arrive)
      DELAY_UNTIL ~no_message(procuid);
      EFFECTS
        msg_queue = EFFECTS_OF read_messages(procuid);

  VFUN h_proc_msg_queue(procuid) -> msg_queue;
      $(returns contents of process message queue)
      HIDDEN;
      INITIALLY msg_queue = VECTOR ();

  OFUN set_real_timer(delta_t; real_timer_id; level)[procuid];
      $(sets real timer to wakeup process in delta_t time units)
      EXCEPTIONS
        invalid_timer(delta_t);
        undefined_level(procuid, level);
      EFFECTS
        'h_real_timer(procuid) = delta_t + read_real_clock();
        'h_real_timer_msg_(procuid) = real_timer_id;
        'h_real_timer_level(procuid) = level;
        'h_proc_real_timer(procuid) = TRUE;

  VFUN h_proc_real_timer(procuid) -> b;
      $(returns true if process has real timer)
      HIDDEN;
      INITIALLY b= FALSE;

  VFUN h_real_timer(procuid) -> time;
      $(returns time of next real timer wakeup for process)
      HIDDEN;
      INITIALLY time = ?;

  VFUN read_real_timer()[procuid] -> time;
      $(external form of h_real_timer)
      EXCEPTIONS
        no_real_timer(procuid);
      DERIVATION h_real_timer(procuid);

  VFUN h_real_timer_msg(procuid) -> real_timer_id;
      $(returns real timer wakeup message)
      HIDDEN;
      INITIALLY real_timer_id = ?;

  VFUN h_real_timer_level(procuid) -> level;
      $(returns level of real timer handler)
      HIDDEN;
```

```
     INITIALLY level = ?;

OFUN real_timer_runout();
     $(notifies processes of real timer runout)
     EFFECTS
       FORALL procuid : h_proc_exists(procuid)
                        AND h_proc_real_timer(procuid)
                        AND h_real_timer(procuid) = read_real_clock() :
         EFFECTS_OF wake(procuid, h_real_timer_msg(procuid)) AND
         'h_proc_real_timer(procuid) = FALSE AND
         EFFECTS_OF set_level(procuid, h_real_timer_level(procuid), TRUE);

VFUN h_virtual_clock(procuid) -> time;
     $(returns the value of process-local virtual time clock)
     HIDDEN;
     INITIALLY time = 0;

OFUN advance_virtual_clock(procuid);
     $(advances virtual time clock one time unit)
     EFFECTS
     'h_virtual_clock(procuid) = h_virtual_clock(procuid) + 1;

VFUN read_virtual_clock(target)[procuid] -> time;
     $(external form of h_virtual_clock)
     EXCEPTIONS
       procuid ~= time_monitor;
       no_process(target);
     DERIVATION h_virtual_clock(target);


END_MODULE
```

```
MODULE  volumes

     TYPES

level_number : {INTEGER ln : 0 <= ln AND ln <= max_ln};
category_set : {VECTOR_OF BOOLEAN cs : LENGTH(cs) = cs_size};
security_level : STRUCT (level_number sln;
                         category_set scs);
integrity_level : STRUCT (level_number iln;
                          category_set ics);
access_level : STRUCT (security_level sl;
                       integrity_level il);
process_uid : INTEGER;
volume_uid : INTEGER;
quota_cell_uid : INTEGER;


     DECLARATIONS

volume_uid voluid;
process_uid procuid;
quota_cell_uid qcuid;
access_level min_al, max_al, al, from_al, to_al;
BOOLEAN b;
INTEGER n;


     PARAMETERS

INTEGER volume_size (number of pages on a volume);
INTEGER initial_disc_drive_count(al) $(initial allocation of drives);
process_uid initializer_id $(volume initializer process id);
INTEGER max_volumes $(maximum number of volumes);
INTEGER max_quota_cell_count $(maximum number of quota cells per volume);


     DEFINITIONS

BOOLEAN write_not_allowed(procuid; al) IS
        ~h_write_allowed(h_proc_trusted(procuid), h_proc_al(procuid), al);

BOOLEAN mounted_volume(voluid) IS
        h_vol_mounted(voluid);

BOOLEAN no_volume(procuid; voluid) IS
        IF ~h_vol_exists(voluid) THEN TRUE
        ELSE ~h_read_allowed(h_proc_trusted(procuid), h_proc_al(procuid),
                             h_vol_min_al(voluid));

BOOLEAN unmounted_volume(voluid) IS
```

```
     IF ~h_vol_mounted(voluid) THEN TRUE
     ELSE ~h_read_allowed(h_proc_trusted(procuid), h_proc_al(procuid),
                          h_vol_min_al(voluid));

BOOLEAN no_disc_drive(al) IS
     h_drive_count(al) = 0;

BOOLEAN invalid_drive_count(n) IS
     n < 0;

BOOLEAN insufficient_drives(al; n) IS
     h_drive_count(al) < n;


    EXTERNALREFS

FROM clock :
    OVFUN get_uid() -> voluid;

FROM access_levels :
    level_number max_ln $(maximum level number);
    INTEGER cs_size $(category set size);
    VFUN h_write_allowed(b; al; al) -> b;
    VFUN h_read_allowed(b; al; al) -> b;

FROM processes :
    VFUN h_proc_al(procuid) -> al;
    VFUN h_proc_trusted(procuid) -> b;

FROM quota_cells :
    OVFUN create_quota_cell(voluid; al; al)[procuid] -> qcuid;
    OFUN set_quota(qcuid; n);
    OFUN set_quota_cell_limit(voluid; n);

FROM address_spaces :
    OFUN revoke_vol_access(voluid; procuid);


    FUNCTIONS

OVFUN create_volume(min_al; max_al)[procuid] -> voluid;
    $(creates a new volume)
    EXCEPTIONS
      procuid ~= initializer_id;
      h_vol_count() = max_volumes;
    EFFECTS
      voluid = EFFECTS_OF get_uid();
      'h_vol_exists(voluid) = TRUE;
      'h_vol_min_al(voluid) = min_al;
      'h_vol_max_al(voluid) = max_al;
      EFFECTS_OF set_quota_cell_limit(voluid, max_quota_cell_count);
    'h_vol_init_qc(voluid) = EFFECTS_OF create_quota_cell(voluid,
                                min_al, min_al, procuid);
    EFFECTS_OF set_quota('h_vol_init_qc(voluid), volume_size);
    'h_vol_count() = h_vol_count() + 1;

OFUN delete_volume(voluid)[procuid];
  $(deletes a volume)
  EXCEPTIONS
    procuid ~= initializer_id;
    ~h_vol_exists(voluid);
    mounted_volume(voluid);
  EFFECTS
    'h_vol_exists(voluid) = FALSE;
    'h_vol_count() = h_vol_count() - 1;

VFUN h_vol_exists(voluid) -> b;
  $(returns true if volume exists)
  HIDDEN;
  INITIALLY b = FALSE;

VFUN h_vol_count() -> n;
  $(returns number of existing volumes)
  HIDDEN;
  INITIALLY n = 0;

VFUN h_vol_min_al(voluid) -> min_al;
  $(returns minimum access level of volume)
  HIDDEN;
  INITIALLY min_al = ?;

VFUN vol_min_al(voluid)[procuid] -> min_al;
  $(external form of h_vol_min_al)
  EXCEPTIONS
    no_volume(procuid, voluid);
  DERIVATION h_vol_min_al(voluid);

VFUN h_vol_max_al(voluid) -> max_al;
  $(returns maximum access level of volume)
  HIDDEN;
  INITIALLY max_al = ?;

VFUN vol_max_al(voluid)[procuid] -> max_al;
  $(external form of h_vol_max_al)
  EXCEPTIONS
    no_volume(procuid, voluid);
  DERIVATION h_vol_max_al(voluid);

VFUN h_vol_init_qc(voluid) -> qcuid;
  $(returns initial quota cell id for volume)
  HIDDEN;
  INITIALLY qcuid = ?;
```

```
VFUN vol_init_qc(voluid)[procuid] -> qcuid;
     $(external form of h_vol_init_qc)
     EXCEPTIONS
       no_volume(procuid, voluid);
     DERIVATION h_vol_init_qc(voluid);

OFUN mount_volume(voluid)[procuid];
     $(mounts a volume on a disc drive)
     EXCEPTIONS
       no_volume(procuid, voluid);
       mounted_volume(voluid);
       write_not_allowed(procuid, h_vol_min_al(voluid));
       no_disc_drive(h_vol_min_al(voluid));
     EFFECTS
       'h_vol_mounted(voluid) = TRUE;
       'h_drive_count(h_vol_min_al(voluid)) =
                         h_drive_count(h_vol_min_al(voluid)) - 1;

OFUN demount_volume(voluid)[procuid];
     $(demounts a volume from a disc drive)
     EXCEPTIONS
       unmounted_volume(procuid, voluid);
       write_not_allowed(procuid, h_vol_min_al(voluid));
     EFFECTS
       'h_vol_mounted(voluid) = FALSE;
       'h_drive_count(h_vol_min_al(voluid)) =
                         h_drive_count(h_vol_min_al(voluid)) + 1;
       EFFECTS_OF revoke_vol_access(voluid, procuid);

VFUN h_vol_mounted(voluid) -> b;
     $(returns true if volume is mounted)
     HIDDEN;
     INITIALLY b = FALSE;

VFUN h_drive_count(al) -> n;
     $(returns number of available disc drives at assigned al)
     HIDDEN;
     INITIALLY n = initial_disc_drive_count(al);

OFUN change_drive_count(from_al; to_al; n)[procuid];
     $(changes access level distribution of free disc drives)
     EXCEPTIONS
       procuid ~= initializer_id;
       invalid_drive_count(n);
       insufficient_drives(from_al, n);
     EFFECTS
       'h_drive_count(from_al) = h_drive_count(from_al) - n;
       'h_drive_count(to_al) = h_drive_count(to_al) + n;
```

```
END_MODULE
```

```
MODULE  quota_cells


     TYPES

level_number : {INTEGER ln : 0 <= ln AND ln <= max_ln};
category_set : {VECTOR_OF BOOLEAN cs : LENGTH(cs) = cs_size};
security_level : STRUCT (level_number sln;
                         category_set scs);
integrity_level : STRUCT (level_number iln;
                          category_set ics);
access_level : STRUCT (security_level sl;
                       integrity_level il);
process_uid : INTEGER;
volume_uid : INTEGER;
quota_cell_uid : INTEGER;


     DECLARATIONS

process_uid procuid;
volume_uid voluid;
quota_cell_uid qcuid, from_qcuid, to_qcuid;
access_level al, val, from_al, to_al;
INTEGER npages, n;
BOOLEAN b;


     DEFINITIONS

BOOLEAN unmounted_volume(procuid; voluid) IS
        IF ~h_vol_mounted(voluid) THEN TRUE
        ELSE ~h_read_allowed(h_proc_trusted(procuid), h_proc_al(procuid),
                             h_vol_min_al(voluid));

BOOLEAN outside_vol_levels(voluid; val; al) IS
        ~h_write_allowed(FALSE, h_vol_min_al(voluid), val) OR
        ~h_write_allowed(FALSE, al, h_vol_max_al(voluid));

BOOLEAN unordered_access_levels(val; al) IS
        ~h_write_allowed(FALSE, val, al);

BOOLEAN read_not_allowed(procuid; al) IS
        ~h_read_allowed(h_proc_trusted(procuid), h_proc_al(procuid), al);

BOOLEAN write_not_allowed(procuid; al) IS
        ~h_write_allowed(h_proc_trusted(procuid), h_proc_al(procuid), al);

BOOLEAN read_write_not_allowed(procuid; al) IS
        ~h_read_write_allowed(h_proc_trusted(procuid), h_proc_al(procuid),
                              al);
```

```
BOOLEAN no_quota_cell(proculd; voluid; qcuid) IS
      IF ~h_qc_exists(voluid, qcuid) THEN TRUE
      ELSE ~h_read_allowed(h_proc_trusted(proculd), h_proc_al(proculd),
                           h_qc_visibility_al(qcuid));

BOOLEAN non_zero_quota(qcuid) IS
      h_qc_pages(qcuid) ~= 0;

BOOLEAN non_zero_refs(qcuid) IS
      h_qc_refs(qcuid) ~= 0;

BOOLEAN invalid_quota_change(npages) IS
      npages < 0;

BOOLEAN insufficient_quota(qcuid; npages) IS
      h_qc_pages(qcuid) - h_qc_pages_used(qcuid) < npages;


    EXTERNALREFS

FROM clock :
     OVFUN get_uid() -> qcuid;

FROM access_levels :
     level_number max_ln $(maximum level number);
     INTEGER cs_size $(category set size);
     VFUN h_read_allowed(b; al; al) -> b;
     VFUN h_write_allowed(b; al; al) -> b;
     VFUN h_read_write_allowed(b; al; al) -> b;

FROM processes :
     VFUN h_proc_al(proculd) -> al;
     VFUN h_proc_trusted(proculd) -> b;

FROM volumes :
     VFUN h_vol_mounted(voluid) -> b;
     VFUN h_vol_min_al(voluid) -> al;
     VFUN h_vol_max_al(voluid) -> al;


    FUNCTIONS

OVFUN create_quota_cell(voluid; val; al)[proculd] -> qcuid;
     $(creates a new quota cell)
     EXCEPTIONS
       unordered_access_levels(val, al);
       write_not_allowed(proculd, val);
       unmounted_volume(proculd, voluid);
       outside_vol_levels(voluid, val, al);
       h_qc_count(voluid, val) = h_qc_limit(voluid, val);
     EFFECTS
       qcuid = EFFECTS_OF get_uid();
       'h_qc_visibility_al(qcuid) = val;
       'h_qc_al(qcuid) = al;
       'h_qc_exists(voluid, qcuid) = TRUE;
       'h_qc_count(voluid, val) = h_qc_count(voluid, val) + 1;

OFUN delete_quota_cell(voluid; qcuid)[procuid];
     $(deletes a quota cell)
     EXCEPTIONS
       unmounted_volume(procuid, voluid);
       no_quota_cell(procuid, voluid, qcuid);
       write_not_allowed(procuid, h_qc_visibility_al(qcuid));
       read_not_allowed(procuid, h_qc_al(qcuid));
       non_zero_quota(qcuid);
       non_zero_refs(qcuid);
     EFFECTS
       'h_qc_exists(voluid, qcuid) = FALSE;
       'h_qc_count(voluid, h_visibility_al(qcuid)) =
                         h_qc_count(voluid, h_visibility_al(qcuid)) - 1;

VFUN h_qc_count(voluid; al) -> n;
     $(returns number of quota cells for volume at given al)
     HIDDEN;
     INITIALLY n = 0;

VFUN h_qc_exists(voluid; qcuid) -> b;
     $(returns true if quota cell exists)
     HIDDEN;
     INITIALLY b = FALSE;

VFUN h_qc_visibility_al(qcuid) -> al;
     $(returns access level of quota cell visibility)
     HIDDEN;
     INITIALLY al = ?;

VFUN qc_visibility_al(voluid; qcuid)[procuid] -> al;
     $(external form of h_qc_visibility_al)
     EXCEPTIONS
       unmounted_volume(procuid, voluid);
       no_quota_cell(procuid, voluid, qcuid);
     DERIVATION h_qc_visibility_al(qcuid);

VFUN h_qc_al(qcuid) -> al;
     $(returns access level of quota cell)
     HIDDEN;
     INITIALLY al = ?;

VFUN qc_al(voluid; qcuid)[procuid] -> al;
     $(external form of h_qc_al)
     EXCEPTIONS
```

```
      unmounted_volume(procuid, voluid);
      no_quota_cell(procuid, voluid, qcuid);
    DERIVATION h_qc_al(qcuid);

OFUN set_quota(qcuid; npages);
    $(sets quota of quota cell
      used only to initialize first quota cell on a volume)
    EFFECTS
      'h_qc_pages(qcuid) = npages;

OFUN move_quota(voluid; from_qcuid; to_qcuid; npages)[procuid];
    $(moves page quota from one quota cell to another)
    EXCEPTIONS
      invalid_quota_change(npages);
      unmounted_volume(procuid, voluid);
      no_quota_cell(procuid, voluid, from_qcuid);
      no_quota_cell(procuid, voluid, to_qcuid);
      read_write_not_allowed(procuid, h_qc_al(from_qcuid));
      write_not_allowed(procuid, h_qc_al(to_qcuid));
      insufficient_quota(from_qcuid, npages);
    EFFECTS
      'h_qc_pages(from_qcuid) = h_qc_pages(from_qcuid) - npages;
      'h_qc_pages(to_qcuid) = h_qc_pages(to_qcuid) + npages;

VFUN h_qc_pages(qcuid) -> npages;
    $(returns page quota for quota cell)
    HIDDEN;
    INITIALLY npages = 0;

VFUN qc_pages(voluid; qcuid)[procuid] -> npages;
    $(external form of h_qc_pages)
    EXCEPTIONS
      unmounted_volume(procuid, voluid);
      no_quota_cell(procuid, voluid, qcuid);
      read_not_allowed(procuid, h_qc_al(qcuid));
    DERIVATION h_qc_pages(qcuid);

VFUN h_qc_refs(qcuid) -> n;
    $(returns quota cell reference count)
    HIDDEN;
    INITIALLY n = 0;

VFUN qc_refs(voluid; qcuid)[procuid] -> n;
    $(external form of h_qc_refs);
    EXCEPTIONS
      unmounted_volume(procuid, voluid);
      no_quota_cell(procuid, voluid, qcuid);
      read_not_allowed(procuid, h_qc_al(qcuid));
    DERIVATION h_qc_refs(qcuid);

OFUN change_qc_refs(qcuid; n);
```

```
     $(changes quota cell reference count)
     EFFECTS
       'h_qc_refs(qcuid) = h_qc_refs(qcuid) + n;

VFUN h_qc_pages_used(qcuid) -> npages;
     $(returns pages used within page quota for quota cell)
     HIDDEN;
     INITIALLY npages = 0;

VFUN qc_pages_used(voluid; qcuid)[procuid] -> npages;
     $(external form of h_qc_pages_used)
     EXCEPTIONS
       unmounted_volume(procuid, voluid);
       no_quota_cell(procuid, voluid, qcuid);
       read_not_allowed(procuid, h_qc_al(qcuid));
     DERIVATION h_qc_pages_used(qcuid);

OFUN change_qc_pages_used(qcuid; npages);
     $(changes pages used for quota cell)
     EFFECTS
       'h_qc_pages_used(qcuid) = h_qc_pages_used(qcuid) + npages;

OFUN set_quota_cell_limit(voluid; n);
     $(sets quota cell limit for volume
       used only to initialize first quota cell limit on a volume)
     EFFECTS
       'h_qc_limit(voluid, h_vol_min_al(voluid)) = n;

VFUN h_qc_limit(voluid; al) -> n;
     $(returns quota cell limit for volume at given al)
     HIDDEN;
     INITIALLY n = 0;

OFUN change_quota_cell_limit(voluid; from_al; to_al; n)[procuid];
     $(moves quota for quota cells from one access level to another)
     EXCEPTIONS
       invalid_quota_change(n);
       unmounted_volume(procuid, voluid);
       read_write_not_allowed(procuid, from_al);
       write_not_allowed(procuid, to_al);
       h_qc_limit(voluid, from_al) - h_qc_count(voluid, from_al) < n;
     EFFECTS
       'h_qc_limit(voluid, from_al) = h_qc_limit(voluid, from_al) - n;
       'h_qc_limit(voluid, to_al) = h_qc_limit(voluid, to_al) + n;


END_MODULE
```

```
MODULE  segments


     TYPES

level_number : {INTEGER ln : 0 <= ln AND ln <= max_ln};
category_set : {VECTOR_OF BOOLEAN cs : LENGTH(cs) = cs_size};
security_level : STRUCT (level_number sln;
                         category_set scs);
integrity_level : STRUCT (level_number iln;
                          category_set ics);
access_level : STRUCT (security_level sl;
                       integrity_level il);
process_uid : INTEGER;
volume_uid : INTEGER;
quota_cell_uid : INTEGER;
segment_uid :INTEGER;
segment_offset : {INTEGER so : 0 <= so AND so <= max_offset};
page_number : {INTEGER pn : 1 <= pn AND pn <= (max_offset + 1)/page_size};
core_address : {INTEGER addr : 0 <= addr AND addr <= max_core_address};
free_page_list : VECTOR_OF core_address;
segment_length : {INTEGER length : 1 <= length AND length <= max_offset
                                                               + 1};
memory_utilization_data : STRUCT (segment_uid seguid;
                                  page_number pageno;
                                  core_address addr;
                                  BOOLEAN used;
                                  BOOLEAN modified);
memory_utilization_data_list : SET_OF memory_utilization_data;
page_data : STRUCT (segment_uid seguid;
                    page_number pageno);
page_data_list : SET_OF page_data;
machine_word : INTEGER;
segment_number : {INTEGER sn : 0 <= sn AND sn <= max_segno};


     DECLARATIONS

process_uid procuid;
volume_uid voluid;
quota_cell_uid qcuid;
segment_uid seguid;
access_level al, val;
segment_offset offset;
page_number pageno;
core_address core_address;
segment_length length;
machine_word word;
free_page_list free_list;
memory_utilization_data mem_data;
```

```
memory_utilization_data_list mem_list;
page_data_list page_list;
segment_number segno;
INTEGER i, page_request_count, npages, n;
BOOLEAN b;


    PARAMETERS

process_uid memory_manager $(memory manager process id);
segment_offset max_offset $(maximum segment offset);
INTEGER page_size $(number of words in a page);
core_address max_core_address $(maximum allowable memory address);
INTEGER min_free_page_count $(minimum number of free pages);
free_page_list init_free_list $(free pages in memory initially);
core_address init_core_address(seguid) $(initial address of unpaged
                                         segments);


    DEFINITIONS

BOOLEAN unmounted_volume(procuid; voluid) IS
      IF ~h_vol_mounted(voluid) THEN TRUE
      ELSE ~h_read_allowed(h_proc_trusted(procuid), h_proc_al(procuid),
                          h_vol_min_al(voluid));

BOOLEAN no_quota_cell(procuid; voluid; qcuid) IS
      IF ~h_qc_exists(voluid, qcuid) THEN TRUE
      ELSE ~h_read_allowed(h_proc_trusted(procuid), h_proc_al(procuid),
                          h_qc_visibility_al(qcuid));

BOOLEAN outside_qc_levels(qcuid; val) IS
      ~h_write_allowed(FALSE, h_qc_visibility_al(qcuid), val) OR
      ~h_write_allowed(FALSE, val, h_qc_al(qcuid));

BOOLEAN write_not_allowed(procuid; al) IS
      ~h_write_allowed(h_proc_trusted(procuid), h_proc_al(procuid), al);

BOOLEAN no_segment(procuid; voluid; seguid) IS
      IF ~h_seg_exists(voluid, seguid) THEN TRUE
      ELSE ~h_read_allowed(h_proc_trusted(procuid), h_proc_al(procuid),
                          h_seg_visibility_al(seguid));

BOOLEAN read_not_allowed(procuid; al) IS
      ~h_read_allowed(h_proc_trusted(procuid), h_proc_al(procuid), al);

BOOLEAN page_quota_overflow(qcuid; npages) IS
      h_qc_pages(qcuid) - h_qc_pages_used(qcuid) < npages;


    EXTERNALREFS
```

```
FROM clock :
    OVFUN get_uid() -> seguid;

FROM access_levels :
    level_number max_ln $(maximum level number);
    INTEGER cs_size $(category set size);
    VFUN h_read_allowed(b; al; al) -> b;
    VFUN h_write_allowed(b; al; al) -> b;

FROM processes :
    VFUN h_proc_exists(procuid) -> b;
    VFUN h_proc_al(procuid) -> al;
    VFUN h_proc_trusted(procuid) -> b;
    OFUN wake(procuid; page_request_count);

FROM volumes :
    VFUN h_vol_min_al(voluid) -> al;
    VFUN h_vol_mounted(voluid) -> b;

FROM quota_cells :
    VFUN h_qc_exists(voluid; qcuid) -> b;
    VFUN h_qc_visibility_al(qcuid) -> al;
    VFUN h_qc_al(qcuid) -> al;
    OFUN change_qc_refs(qcuid; i);
    OFUN change_qc_pages_used(qcuid; i);
    VFUN h_qc_pages(qcuid) -> npages;
    VFUN h_qc_pages_used(qcuid) -> npages;

FROM address_spaces :
    segment_number max_segno $(maximum segment number);
    VFUN h_kst_seguid(procuid; segno) -> seguid;
    VFUN h_kst_seg_io_count(procuid; segno) -> i;
    VFUN h_kst_valid(procuid; segno) -> b;


    FUNCTIONS

OVFUN create_seg(voluid; qcuid; val; length)[procuid] -> seguid;
    $(creates a new segment)
    DEFINITIONS
      INTEGER n IS (length - 1 + page_size)/page_size;
    EXCEPTIONS
      unmounted_volume(procuid, voluid)
      no_quota_cell(procuid, voluid, qcuid);
      outside_qc_levels(qcuid, val);
      write_not_allowed(procuid, val);
      page_quota_overflow(qcuid, n);
    EFFECTS
      seguid = EFFECTS_OF get_uid();
      'h_seg_exists(voluid,seguid) = TRUE;
      'h_seg_qc(seguid) = qcuid;
      'h_seg_visibility_al(seguid) = val;
      EFFECTS_OF change_qc_refs(qcuid, 1);
      EFFECTS_OF change_qc_pages_used(qcuid, n);
      'h_seg_length(seguid) = length;
      'h_seg_paged(seguid) = TRUE;

OFUN delete_seg(voluid; seguid)[procuid];
    $(deletes a segment)
    EXCEPTIONS
      unmounted_volume(procuid, voluid);
      no_segment(procuid, voluid, seguid);
      write_not_allowed(procuid, h_seg_visibility_al(seguid));
    DELAY_UNTIL FORALL proc : h_proc_exists(proc) :
                (FORALL segno : h_kst_valid(proc, segno) AND
                                h_kst_seguid(proc, segno) = seguid :
                 h_kst_seg_io_count(proc, segno) = 0);
    DEFINITIONS
      INTEGER n IS (h_seg_length(seguid) - 1 + page_size)/page_size;
      quota_cell_uid qcuid IS h_seg_qc(seguid);
    EFFECTS
      'h_seg_exists(voluid, seguid) = FALSE;
      EFFECTS_OF change_qc_refs(qcuid, -1);
      EFFECTS_OF change_qc_pages_used(qcuid, -n);
      IF h_seg_paged(seguid)
      THEN FORALL i : i >= 1 AND i <= n :
           EFFECTS_OF deallocate_page(seguid, i)
      ELSE FORALL i : i >= 0 AND i < n :
           EFFECTS_OF add_page_to_free_list(h_seg_core_address(seguid)
                                            + i*page_size);

OVFUN read(seguid; offset) -> word;
    $(reads a word of a segment)
    DEFINITIONS
      INTEGER n IS (offset + page_size)/page_size;
    EFFECTS
      h_seg_paged(seguid) => EFFECTS_OF allocate_page(seguid, n) AND
                             'h_seg_page_used(seguid, n) = TRUE;
      word = h_seg_contents(seguid, offset);

OFUN write(seguid; offset; word);
    $(writes a word of a segment)
    DEFINITIONS
      INTEGER n IS (offset + page_size)/page_size;
    EFFECTS
      h_seg_paged(seguid) => EFFECTS_OF allocate_page(seguid, n) AND
                             'h_seg_page_used(seguid, n) = TRUE AND
                             'h_seg_page_modified(seguid, n) = TRUE;
      'h_seg_contents(seguid, offset) = word;

OFUN allocate_page(seguid; pageno);
```

```
      $(allocates main memory page to segment)
      EFFECTS
        ~h_seg_page_in_core(seguid, pageno) =>
        'h_seg_page_core_address(seguid, pageno) =
                 EFFECTS_OF remove_page_from_free_list() AND
        'h_seg_page_in_core(seguid, pageno) = TRUE;

  OFUN deallocate_page(seguid; pageno);
      $(deallocates segment page from main memory)
      EFFECTS
        h_seg_page_in_core(seguid, pageno) =>
        EFFECTS_OF add_page_to_free_list(h_seg_page_core_address(
                                          seguid, pageno))
        AND 'h_seg_page_in_core(seguid, pageno) = FALSE;

  VFUN h_seg_exists(voluid; seguid) -> b;
      $(returns true if segment exists)
      HIDDEN;
      INITIALLY b = FALSE;

  VFUN h_seg_visibility_al(seguid) -> al;
      $(returns access level of segment creator)
      HIDDEN;
      INITIALLY al = ?;

  VFUN seg_visibility_al(voluid; seguid)[procuid] -> al;
      $(external form of h_seg_visibility_al)
      EXCEPTIONS
        unmounted_volume(procuid, voluid);
        no_segment(procuid, voluid, seguid);
      DERIVATION h_seg_visibility_al(seguid);

  VFUN h_seg_al(seguid) -> al;
      $(returns access level of segment)
      HIDDEN;
      DERIVATION h_qc_al(h_seg_qc(seguid));

  VFUN seg_al(voluid; seguid)[procuid] -> al;
      $(external form of h_seg_al)
      EXCEPTIONS
        unmounted_volume(procuid, voluid);
        no_segment(procuid, voluid, seguid);
      DERIVATION h_seg_al(seguid);

  VFUN h_seg_qc(seguid) -> qcuid;
      $(returns quota cell id of segment)
      HIDDEN;
      INITIALLY qcuid = ?;

  VFUN seg_qc(voluid; seguid)[procuid] -> qcuid;
      $(external form of h_seg_qc)
```

```
     EXCEPTIONS
       unmounted_volume(procuid, voluid);
       no_segment(procuid, voluid, seguid);
     DERIVATION h_seg_qc(seguid);

VFUN h_seg_contents(seguid; offset) -> word;
     $(returns a word of a segment)
     HIDDEN;
     INITIALLY word = 0;

VFUN h_seg_length(seguid) -> length;
     $(returns length of segment)
     HIDDEN;
     INITIALLY length = ?;

VFUN seg_length(voluid; seguid)[procuid] -> length;
     $(external form of h_seg_length)
     EXCEPTIONS
       unmounted_volume(procuid, voluid);
       no_segment(procuid, voluid, seguid);
     DERIVATION h_seg_length(seguid);

VFUN h_seg_paged(seguid) -> b;
     $(returns true if segment is paged)
     HIDDEN;
     INITIALLY b = FALSE;

VFUN h_seg_page_in_core(seguid; pageno) -> b;
     $(returns true if page is in core)
     HIDDEN;
     INITIALLY b = FALSE;

VFUN h_seg_page_core_address(seguid; pageno) -> core_address;
     $(returns core address of page)
     HIDDEN;
     INITIALLY core_address = ?;

VFUN h_seg_core_address(seguid) -> core_address;
     $(returns core address of unpaged segment)
     HIDDEN;
     INITIALLY core_address = init_core_address(seguid);

OFUN set_page_used_indicator(seguid; pageno);
     $(sets global page used indicator)
     EFFECTS
       'h_seg_page_used(seguid, pageno) = TRUE;

OFUN set_page_indicators(seguid; pageno);
     $(sets global page used and modified indicators)
     EFFECTS
       'h_seg_page_used(seguid, pageno) = TRUE;
```

```
        'h_seg_page_modified(seguid, pageno) = TRUE;

 VFUN h_seg_page_used(seguid; pageno) -> b;
      $(returns true if page globally used)
      HIDDEN;
      INITIALLY b = FALSE;

 VFUN h_seg_page_modified(seguid; pageno) -> b;
      $(returns true if page globally modified)
      HIDDEN;
      INITIALLY b = FALSE;

 VFUN h_seg_wire_count(seguid) -> n;
      $(returns count of processes having segment wired)
      HIDDEN;
      INITIALLY n = 0;

 OFUN change_wire_count(seguid; n)
      $(changes wire count for segment)
      EFFECTS
        'h_seg_wire_count(seguid) = h_seg_wire_count(seguid) + n;
        h_seg_wire_count(seguid) = 0 AND h_seg_paged(seguid) =>
          FORALL i : 1 <= i AND i <= (h_seg_length(seguid) - 1
                                      + page_size)/page_size :
            EFFECTS_OF allocate_page(seguid, i);

 OVFUN get_memory_data()[procuid] -> mem_list;
      $(returns memory utilization data to memory manager)
      EXCEPTIONS
        procuid ~= memory_manager;
      DEFINITIONS
        INTEGER n IS (h_seg_length(seguid) - 1 + page_size)/page_size;
      EFFECTS
        mem_list = {memory_utilization_data mem_data :
                    h_vol_mounted(voluid) AND
                    h_seg_exists(voluid, seguid) AND
                    h_seg_paged(seguid) AND
                    h_seg_wire_count(seguid) = 0 AND
                    h_seg_page_in_core(seguid, pageno)};
        FORALL voluid : h_vol_mounted(voluid) :
        (FORALL seguid : h_seg_exists(voluid, seguid) AND
                         h_seg_paged(seguid) AND
                         h_seg_wire_count(seguid) = 0 :
        (FORALL pageno : pageno <= n AND
                         h_seg_page_in_core(seguid, pageno) :
          'h_seg_page_used(seguid, pageno) = FALSE));

 VFUN h_mem_data(seguid; pageno) -> mem_data;
      $(returns memory utilization data)
      HIDDEN;
      DERIVATION <seguid, pageno,
                h_seg_page_core_address(seguid, pageno),
                h_seg_page_used(seguid, pageno),
                h_seg_page_modified(seguid, pageno)>;

OFUN provide_free_pages(page_list)[procuid];
     $(provides free pages from memory manager for kernel free list)
     EXCEPTIONS
       procuid ~= memory_manager;
       page_list ~INSET mem_list;
     DEFINITIONS
       INTEGER n IS CARDINALITY (page_list);
     EFFECTS
       FOR i FROM 1 TO n :
         'h_seg_page_modified(page_list.seguid[i],
                              page_list.pageno[i]) = FALSE
         AND 'h_seg_page_in_core(page_list.seguid[i],
                                 page_list.pageno[i]) = FALSE
         AND EFFECTS_OF add_page_to_free_list(h_seg_page_core_address(
                                              page_list.seguid[i],
                                              page_list.pageno[i]));

VFUN h_free_page_list() -> free_list;
     $(returns list of free pages in main memory)
     HIDDEN;
     INITIALLY free_list = init_free_list

OFUN add_page_to_free_list(core_address);
     $(adds page to list of free pages in main memory)
     DEFINITIONS
       INTEGER n IS LENGTH(h_free_page_list());
     EFFECTS
       'h_free_page_list() = VECTOR (FOR i FROM 1 TO n+1 :
         IF i <= n THEN h_free_page_list()[i] ELSE core_address);

OVFUN remove_page_from_free_list() -> core_address;
     $(removes page from list of free pages in main memory)
     DEFINITIONS
       INTEGER n IS LENGTH(h_free_page_list());
     DELAY_UNTIL n >= min_free_page_count;
     EFFECTS
       'h_free_page_list() = VECTOR (FOR i FROM 1 TO n-1 :
         h_free_page_list()[i];
       core_address = h_free_page_list()[n];
       n = min_free_page_count =>
             EFFECTS_OF wake(memory_manager, page_request_count);


END_MODULE
```

```
MODULE  devices

    TYPES

level_number : {INTEGER ln : 0 <= ln AND ln <= max_ln};
category_set : {VECTOR_OF BOOLEAN cs : LENGTH(cs) = cs_size};
security_level : STRUCT (level_number sln;
                         category_set scs);
integrity_level : STRUCT (level_number iln;
                          category_set ics);
access_level : STRUCT (security_level sl;
                       integrity_level il);
process_uid : INTEGER;
device_uid : INTEGER;
device_mapping_type : {mapped, premapped};
device_type : {INTEGER type : 1 <= type AND type <= max_dev_types};
device_utilization_data : STRUCT (device_uid devuid;
                                  BOOLEAN initiated;
                                  BOOLEAN used;
                                  BOOLEAN modified);
device_utilization_data_list : SET_OF device_utilization_data;
segment_number : {INTEGER sn : 0 <= sn AND sn <= max_segno};
device_number : {INTEGER dn : 0 <= dn AND dn <= max_devno};
machine_word : INTEGER;
control_operation : INTEGER;


    DECLARATIONS

process_uid procuid;
access_level al;
machine_word word;
device_uid devuid;
device_mapping_type map_type;
device_type dev_type;
device_utilization_data dev_data;
device_utilization_data_list dev_list;
device_number devno;
control_operation opcode;
segment_number segno;
INTEGER n;
BOOLEAN b;


    PARAMETERS

process_uid device_manager $(device manager process id);
process_uid device_monitor $(device monitor process id);
device_type max_device_types $(maximum device types);
BOOLEAN valid_read_op(dev_type; opcode) $(valid read ops for device);
BOOLEAN valid_write_op(dev_type; opcode) $(valid write ops for device);
INTEGER max_devices $(maximum number of devices);


     DEFINITIONS

BOOLEAN existing_device(devuid) IS
        h_dev_exists(devuid);

BOOLEAN no_device(procuid; devuid) IS
        IF ~h_dev_exists(devuid) THEN TRUE
        ELSE ~h_read_allowed(h_proc_trusted(procuid), h_proc_al(procuid),
                             h_dev_al(devuid));

BOOLEAN device_not_active(devuid) IS
        ~h_dev_active(devuid);


     EXTERNALREFS

FROM access_levels :
     level_number max_ln $(maximum level number);
     INTEGER cs_size $(category set size);
     VFUN h_read_allowed(b; al; al) -> b;

FROM processes :
     VFUN h_proc_al(procuid) -> al;
     VFUN h_proc_trusted(procuid) -> b;
     OFUN wake(procuid; devno);

FROM address_spaces :
     segment_number max_segno $(maximum segment number);
     device_number max_devno $(maximum device number);
     VFUN h_kdt_dev_used(procuid; devno) -> b;
     VFUN h_kdt_dev_modified(procuid; devno) -> b;
     OFUN reset_device_used_indicator(devuid);
     OFUN reset_device_modified_indicator(devuid);
     VFUN h_kdt_segno(procuid; devno) -> segno;
     OFUN set_level(procuid; level; b);
     OFUN decrement_seg_io_count(procuid; segno);
     OFUN revoke_device_access(devuid)[procuid];
     VFUN h_kdt_level(procuid; devno) -> level;


     FUNCTIONS

OFUN create_device(devuid; al; dev_type; map_type)[procuid];
     $(creates a new device)
     EXCEPTIONS
       procuid ~= device_manager;
       existing_device(devuid);
```

```
      h_dev_count() = max_devices;
    EFFECTS
      'h_dev_exists(devuid) = TRUE;
      'h_dev_al(devuid) = al;
      'h_dev_type(devuid) = dev_type;
      'h_dev_map_type(devuid) = map_type;
      'h_dev_count() = h_dev_count() + 1;

OFUN delete_device(devuid)[procuid];
    $(deletes a device)
    EXCEPTIONS
      procuid ~= device_manager;
      no_device(procuid, devuid);
    EFFECTS
      'h_dev_exists(devuid) = FALSE;
      'h_dev_count() = h_dev_count() - 1;

OFUN change_device_al(devuid; al)[procuid];
    $(changes access level of device)
    EXCEPTIONS
      procuid ~= device_manager;
      no_device(procuid, devuid);
    EFFECTS
      'h_dev_al(devuid) = al;
      EFFECTS_OF revoke_device_access(devuid, procuid);

OFUN change_device_mapping_type(devuid; map_type)[procuid];
    $(changes device mapping type)
    EXCEPTIONS
      procuid ~= device_manager;
      no_device(procuid, devuid);
    EFFECTS
      'h_dev_map_type(devuid) = map_type;

OVFUN get_device_data()[procuid] -> dev_list;
    $(returns device utilization data to device monitor)
    EXCEPTIONS
      procuid ~= device_monitor;
    EFFECTS
      dev_list = {device_utilization_data dev_data :
                  h_dev_exists(devuid)};
      FORALL devuid : h_dev_exists(devuid) AND
                      h_dev_initiated(devuid) :
        EFFECTS_OF reset_device_used_indicator(devuid) AND
        EFFECTS_OF reset_device_modified_indicator(devuid);

VFUN h_dev_data(devuid) -> dev_data;
    $(returns device utilization data)
    HIDDEN;
    DERIVATION <devuid, h_dev_initiated(devuid),
                h_dev_used(devuid), h_dev_modified(devuid)>;
```

```
VFUN h_dev_used(devuid) -> b;
     $(returns device used data)
     HIDDEN;
     DERIVATION IF h_dev_initiated(devuid)
                THEN h_kdt_dev_used(h_dev_initiator(devuid),
                                    h_dev_devno(devuid))
                ELSE FALSE;

VFUN h_dev_modified(devuid) -> b;
     $(returns device modified data)
     HIDDEN;
     DERIVATION IF h_dev_initiated(devuid)
                THEN h_kdt_dev_modified(h_dev_initiator(devuid),
                                        h_dev_devno(devuid))
                ELSE FALSE;

OFUN device_wakeup(devuid);
     $(notifies initiator process of asynchronous io termination)
     EXCEPTIONS
       device_not_active(devuid);
     DEFINITIONS
       process_uid proc IS h_dev_initiator(devuid);
       device_number devno IS h_dev_devno(devuid);
       segment_number segno IS h_kdt_segno(proc, devno);
     EFFECTS
       'h_dev_active(devuid) = FALSE;
       EFFECTS_OF wake(proc, devno);
       EFFECTS_OF set_level(proc, h_kdt_level(proc, devno), TRUE);
       EFFECTS_OF decrement_seg_io_count(proc, segno);

OFUN assign_device(procuid; devuid; devno);
     $(assigns device to initiator process)
     EFFECTS
       'h_dev_initiated(devuid) = TRUE;
       'h_dev_initiator(devuid) = procuid;
       'h_dev_devno(devuid) = devno;

OFUN release_device(devuid);
     $(releases initiated device)
     EFFECTS
       'h_dev_initiated(devuid) = FALSE;

OFUN set_device_active(devuid);
     $(sets device state active)
     EFFECTS
       'h_dev_active(devuid) = TRUE;

OFUN write_device(devuid; word);
     $(writes a word to device)
     EFFECTS
```

```
    'h_dev_contents(devuid) = word;

VFUN h_dev_count() -> n;
    $(returns number of devices)
    HIDDEN;
    INITIALLY n = 0;

VFUN h_dev_exists(devuid) -> b;
    $(returns true if device exists)
    HIDDEN;
    INITIALLY b = FALSE;

VFUN h_dev_al(devuid) -> al;
    $(returns access level of device)
    HIDDEN;
    INITIALLY al = ?;

VFUN h_dev_type(devuid) -> dev_type;
    $(returns device type)
    HIDDEN;
    INITIALLY dev_type = ?;

VFUN h_dev_map_type(devuid) -> map_type;
    $(returns device mapping type)
    HIDDEN;
    INITIALLY map_type = ?;

VFUN h_valid_read_op(dev_type; opcode) -> b;
    $(returns true if valid read operation for device type)
    HIDDEN;
    INITIALLY b = valid_read_op(dev_type, opcode);

VFUN h_valid_write_op(dev_type; opcode) -> b;
    $(returns true if valid write operation for device type)
    HIDDEN;
    INITIALLY b = valid_write_op(dev_type, opcode);

VFUN h_dev_initiated(devuid) -> b;
    $(returns true if device currently initiated)
    HIDDEN;
    INITIALLY b = FALSE;

VFUN h_dev_initiator(devuid) -> procuid;
    $(returns device initiator process id)
    HIDDEN;
    INITIALLY procuid = ?;

VFUN h_dev_devno(devuid) -> devno;
    $(returns local device number in initiator process space)
    HIDDEN;
    INITIALLY devno = ?;
```

```
VFUN h_dev_active(devuid) -> b;
     $(returns true if asynchronous read or write in progress)
     HIDDEN;
     INITIALLY b = FALSE;

VFUN h_dev_contents(devuid) -> word;
     $(returns contents of device)
     HIDDEN;
     INITIALLY word = ?;

VFUN device_al(devuid)[procuid] -> al;
     $(external form of h_dev_al)
     EXCEPTIONS
       no_device(procuid, devuid);
     DERIVATION h_dev_al(devuid);

VFUN device_type(devuid)[procuid] -> dev_type;
     $(external form of h_dev_type)
     EXCEPTIONS
       no_device(procuid, devuid);
     DERIVATION h_dev_type(devuid);

VFUN device_map_type(devuid)[procuid] -> map_type;
     $(external form of h_dev_map_type)
     EXCEPTIONS
       no_device(procuid, devuid);
     DERIVATION h_dev_map_type(devuid);


END_MODULE
```

MODULE address_spaces

```
    TYPES

level_number : {INTEGER ln : 0 <= ln AND ln <= max_ln};
category_set : {VECTOR_OF BOOLEAN cs : LENGTH(cs) = cs_size};
security_level : STRUCT (level_number sln;
                         category_set scs);
integrity_level : STRUCT (level_number iln;
                          category_set ics);
access_level : STRUCT (security_level sl;
                       integrity_level il);
process_uid : INTEGER;
volume_uid : INTEGER;
segment_uid : INTEGER;
ring_number : {INTEGER rn : 0 <= rn AND rn <= max_ring};
ring_brackets : {VECTOR_OF ring_number rb : LENGTH(rb) = 3};
segment_number : {INTEGER sn : 0 <= sn AND sn <= max_segno};
device_number : {INTEGER dn : 0 <= dn AND dn <= max_devno};
segment_offset : {INTEGER so : 0 <= so AND so <= max_offset};
execution_point : STRUCT (ring_number rn;
                          segment_number sn;
                          segment_offset so);
access_mode : {VECTOR_OF BOOLEAN am : LENGTH(am) = 3};
machine_word : INTEGER;
page_number : {INTEGER pn : 1 <= pn AND pn <= (max_offset + 1)/page_size};
level : {INTEGER level : 1 <= level AND level <= pl_size};
process_levels : {VECTOR_OF BOOLEAN pl : LENGTH(pl) = pl_size};
device_uid : INTEGER;
segment_length : {INTEGER length : 1 <= length AND length <= max_offset
                                                                + 1};
device_mapping_type : {mapped, premapped};
trap_number : {INTEGER tn : 1 <= tn AND tn <= max_trapno};
control_operation : INTEGER;
mem_area : STRUCT (segment_number sn;
                   segment_offset so);
io_count : {INTEGER count : 0 <= count AND count <= max_io_count};
trap_info : VECTOR_OF machine_word;
device_type : {INTEGER type : 1 <= type AND type <= max_dev_types};
known_segment_table : STRUCT (BOOLEAN valid;
                              segment_uid seguid;
                              volume_uid voluid;
                              BOOLEAN access_defined;
                              access_mode mode;
                              ring_brackets rb;
                              BOOLEAN seg_used;
                              BOOLEAN seg_modified;
                              BOOLEAN seg_wired;
                              INTEGER seg_io_count;
                              BOOLEAN page_used;
```

```
                            BOOLEAN page_modified);
known_segment_table_list : SET_OF known_segment_table;
```

DECLARATIONS

```
process_uid procuid, proc;
volume_uid voluid;
segment_uid seguid;
device_uid devuid;
access_level al;
access_mode mode, given_mode;
ring_number ring, eff_ring;
ring_brackets rb;
segment_number segno;
device_number devno;
segment_offset offset;
machine_word word;
trap_number trapno;
execution_point exec_pt, return_exec_pt;
page_number pageno;
level level;
process_levels activity_levels;
control_operation opcode;
device_mapping_type map_type;
device_type dev_type;
mem_area save_area;
segment_length length, range;
io_count io_count;
trap_info trap_info;
INTEGER i,n;
BOOLEAN b;
```

PARAMETERS

```
ring_number kernel_ring $(least privileged ring used by kernel);
ring_number max_ring $(maximum ring number);
segment_number max_segno $(maximum segment number);
device_number max_devno $(maximum device number);
trap_number max_trapno $(maximum trap number);
io_count max_io_count $(maximum number of active io operations);
level init_level $(initial process level);
execution_point init_exec_pt $(initial process execution point);
known_segment_table_list template_kst $(initial process kst);
INTEGER pl_size $(number of process levels);
```

DEFINITIONS

```
BOOLEAN unmounted_volume(procuid; voluid) IS
```

```
        IF ~h_vol_mounted(voluid) THEN TRUE
        ELSE ~h_read_allowed(h_proc_trusted(procuid), h_proc_al(procuid),
                             h_vol_min_al(voluid));

BOOLEAN no_segment(procuid; voluid; seguid) IS
        IF ~h_seg_exists(voluid, seguid) THEN TRUE
        ELSE ~h_read_allowed(h_proc_trusted(procuid), h_proc_al(procuid),
                             h_seg_visibility_al(seguid));

BOOLEAN write_not_allowed(procuid; al) IS
        ~h_write_allowed(h_proc_trusted(procuid), h_proc_al(procuid), al);

BOOLEAN invalid_segno(procuid; segno) IS
        ~h_kst_valid(procuid, segno);

BOOLEAN undefined_access(procuid; segno) IS
        ~h_kst_access_defined(procuid, segno);

BOOLEAN no_read_permission(procuid; ring; segno) IS
        ~(h_kst_mode(procuid, segno)[1] AND
          ring <= h_kst_rb(procuid, segno)[2]);

BOOLEAN no_write_permission(procuid; ring; segno) IS
        ~(h_kst_mode(procuid, segno)[2] AND
          ring <= h_kst_rb(procuid, segno)[1]);

BOOLEAN no_execute_permission(procuid; ring; segno) IS
        ~(h_kst_mode(procuid, segno)[3] AND
          ring <= h_kst_rb(procuid, segno)[2] AND
          ring >= h_kst_rb(procuid, segno)[1]);

BOOLEAN out_of_bounds(procuid; segno; offset) IS
        offset >= h_seg_length(h_kst_seguid(procuid, segno));

BOOLEAN outside_call_brackets(procuid; ring; segno) IS
        ~(ring >= h_kst_rb(procuid, segno)[1] AND
          ring <= h_kst_rb(procuid, segno)[3]);

BOOLEAN invalid_call_limiter(offset) IS
        ~(offset = 0);

BOOLEAN inward_return(procuid; ring) IS
        ring < h_proc_ring(procuid);

BOOLEAN segno_in_use(procuid; segno) IS
        h_kst_valid(procuid, segno);

BOOLEAN segment_wired(procuid; segno) IS
        h_kst_seg_wired(procuid, segno);

BOOLEAN invalid_ring_brackets(rb) IS
```

```
       ~(kernel_ring < rb[1] AND
         rb[1] <= rb[2] <= rb[3]);

BOOLEAN segment_not_wired(procuid; segno) IS
       ~h_kst_seg_wired(procuid, segno);

BOOLEAN segment_used_for_io(procuid; segno) IS
       ~(h_kst_seg_io_count(procuid, segno) = 0);

BOOLEAN undefined_level(procuid; level) IS
       ~h_level_handler_exists(procuid, level);

BOOLEAN devno_in_use(procuid; devno) IS
       h_kdt_valid(procuid, devno);

BOOLEAN no_device(procuid; devuid) IS
       IF ~h_dev_exists(devuid) THEN TRUE
       ELSE ~h_read_allowed(h_proc_trusted(procuid), h_proc_al(procuid),
                            h_dev_al(devuid));

BOOLEAN device_in_use(devuid) IS
       h_dev_initiated(devuid);

BOOLEAN invalid_devno(procuid; devno) IS
       ~h_kdt_valid(procuid, devno);

BOOLEAN device_active(procuid; devno) IS
       h_dev_active(h_kdt_devuid(procuid, devno));

BOOLEAN invalid_device_ring_brackets(rb) IS
       ~(kernel_ring < MIN([rb[1], rb[3]]) AND rb[1] <= rb[2]);

BOOLEAN invalid_device_read(procuid; ring; devuid; devno; opcode) IS
       ~((h_kdt_mode(procuid, devno)[1] AND
          ring <= h_kdt_rb(procuid, devno)[2] AND
          h_valid_read_op(h_dev_type(devuid), opcode)) OR
         (h_kdt_mode(procuid, devno)[3] AND
          ring <= h_kdt_rb(procuid, devno)[3]));

BOOLEAN invalid_device_write(procuid; ring; devuid; devno; opcode) IS
       ~((h_kdt_mode(procuid, devno)[2] AND
          ring <= h_kdt_rb(procuid, devno)[1] AND
          h_valid_write_op(h_dev_type(devuid), opcode)) OR
         (h_kdt_mode(procuid, devno)[3] AND
          ring <= h_kdt_rb(procuid, devno)[3]));

BOOLEAN undefined_device_access(procuid; devno) IS
       ~h_kdt_access_defined(procuid, devno);

BOOLEAN no_device_write_permission(procuid; ring; devno) IS
       ~(h_kdt_mode(procuid, devno)[2] AND
          ring <= h_kdt_rb(procuid, devno)[1]);

BOOLEAN no_device_read_permission(procuid; ring; devno) IS
        ~(h_kdt_mode(procuid, devno)[1] AND
          ring <= h_kdt_rb(procuid, devno)[2]);

BOOLEAN max_io_operations(procuid; segno) IS
        h_kst_seg_io_count(procuid, segno) = max_io_count;

BOOLEAN invalid_memory_limits(procuid; devuid; seguid; offset; range) IS
        IF h_dev_map_type(devuid) = mapped
        THEN offset >= h_seg_length(seguid)
        ELSE (offset + range) >= h_seg_length(seguid) OR
             (h_seg_paged(seguid) AND
              (offset MOD page_size + range) > page_size);

BOOLEAN no_write_memory_permission(procuid; devuid; ring; segno) IS
        IF h_dev_map_type(devuid) = premapped
        THEN ~(h_kst_mode(procuid, segno)[2] AND
               ring <= h_kst_rb(procuid, segno)[1])
        ELSE FALSE;

BOOLEAN no_read_memory_permission(procuid; devuid; ring; segno) IS
        IF h_dev_map_type(devuid) = premapped
        THEN ~(h_kst_mode(procuid, segno)[1] AND
               ring <= h_kst_rb(procuid, segno)[2])
        ELSE FALSE;

BOOLEAN device_not_active(devuid) IS
        ~h_dev_active(devuid);

BOOLEAN invalid_segment_reference(procuid; devno; segno) IS
        ~(h_kdt_segno(procuid, devno) = segno);

BOOLEAN existing_trap_handler(procuid; trapno; ring) IS
        (h_trap_handler_exists(procuid, trapno) AND
         ring > h_trap_ring(procuid, trapno));

BOOLEAN no_trap_handler(procuid; trapno) IS
        ~h_trap_handler_exists(procuid, trapno);

BOOLEAN no_trap_outstanding(procuid; trapno) IS
        h_trap_depth(procuid, trapno) = 0;

BOOLEAN inner_ring_handler(procuid; ring) IS
        h_proc_ring(procuid) > ring;

BOOLEAN existing_level_handler(procuid; level; ring) IS
        (h_level_handler_exists(procuid, level) AND
         ring > h_level_ring(procuid, level));
```

```
     EXTERNALREFS

FROM access_levels :
     level_number max_ln $(maximum level number);
     INTEGER cs_size $(category set size);
     VFUN h_read_allowed(b; al; al) -> b;
     VFUN h_write_allowed(b; al; al) -> b;
     VFUN h_read_write_allowed(b; al; al) -> b;

FROM processes :
     VFUN h_proc_exists(procuid) -> b;
     VFUN h_proc_al(procuid) -> al;
     VFUN h_proc_trusted(procuid) -> b;

FROM volumes :
     VFUN h_vol_min_al(voluid) -> al;
     VFUN h_vol_mounted(voluid) -> b;

FROM segments :
     segment_offset max_offset $(maximum segment offset);
     INTEGER page_size $(number of words in a page);
     VFUN h_seg_exists(voluid; seguid) -> b;
     VFUN h_seg_visibility_al(seguid) -> al;
     VFUN h_seg_al(seguid) -> al;
     VFUN h_seg_length(seguid) -> length;
     VFUN h_seg_contents(seguid; offset) -> word;
     VFUN h_seg_paged(seguid) -> b;
     VFUN h_seg_page_in_core(seguid; pageno) -> b;
     OVFUN read(seguid; offset) -> word;
     OFUN write(seguid; offset; word);
     OFUN allocate_page(seguid; pageno);
     OFUN change_wire_count(seguid; n);
     OFUN set_page_used_indicator(seguid; pageno);
     OFUN set_page_indicators(seguid; pageno);

FROM devices :
     device_type max_dev_types $(maximum device types);
     VFUN h_dev_exists(devuid) -> b;
     VFUN h_dev_al(devuid) -> al;
     VFUN h_dev_type(devuid) -> dev_type;
     VFUN h_dev_map_type(devuid) -> map_type;
     VFUN h_valid_read_op(dev_type; opcode) -> b;
     VFUN h_valid_write_op(dev_type; opcode) -> b;
     VFUN h_dev_initiated(devuid) -> b;
     VFUN h_dev_initiator(devuid) -> procuid;
     VFUN h_dev_devno(devuid) -> devno;
     VFUN h_dev_active(devuid) -> b;
     VFUN h_dev_contents(devuid) -> word;
     OFUN assign_device(procuid; devuid; devno);
     OFUN release_device(devuid);
```

```
      OFUN set_device_active(devuid);
      OFUN write_device(devuid; word);


      FUNCTIONS

 OFUN assign_segno(voluid; seguid; segno)[procuid];
      $(assigns a segment number to a segment for a process)
      EXCEPTIONS
        segno_in_use(procuid, segno);
        unmounted_volume(procuid, voluid);
        no_segment(procuid, voluid, seguid);
      EFFECTS
        'h_kst_valid(procuid, segno) = TRUE;
        'h_kst_voluid(procuid, segno) = voluid;
        'h_kst_seguid(procuid, segno) = seguid;
        'h_kst_access_defined(procuid, segno) = FALSE;
        'h_kst_seg_wired(procuid, segno) = FALSE;
        'h_kst_seg_io_count(procuid, segno) = 0;
        ~h_seg_paged(seguid) =>
             'h_kst_seg_used(procuid, segno) = FALSE AND
             'h_kst_seg_modified(procuid, segno) = FALSE;

 OFUN release_segno(segno)[procuid];
      $(releases a segment number for a process)
      EXCEPTIONS
        invalid_segno(procuid, segno);
        segment_wired(procuid, segno);
      EFFECTS
        'h_kst_valid(procuid, segno) = FALSE;

 OVFUN give_access(segno; mode; rb)[procuid] -> given_mode;
      $(gives a process access to a segment and returns given mode)
      DEFINITIONS
        volume_uid voluid IS h_kst_voluid(procuid, segno);
        segment_uid seguid IS h_kst_seguid(procuid, segno);
        access_level al IS h_seg_al(seguid);
        BOOLEAN readable IS h_read_allowed(h_proc_trusted(procuid),
                                           h_proc_al(procuid), al);
        BOOLEAN writable IS h_write_allowed(h_proc_trusted(procuid),
                                            h_proc_al(procuid), al);
        access_mode max_mode IS VECTOR(readable, writable, readable);
      EXCEPTIONS
        invalid_ring_brackets(rb);
        invalid_segno(procuid, segno);
        unmounted_volume(procuid, voluid);
        no_segment(procuid, voluid, seguid);
      EFFECTS
        given_mode = VECTOR (FOR i FROM 1 TO 3 :
                          mode[i] AND max_mode[i]);
        'h_kst_mode(procuid, segno) = given_mode;
        'h_kst_rb(procuid, segno) = rb;
        'h_kst_access_defined(procuid, segno) = TRUE;

OFUN revoke_access(voluid; seguid)[procuid];
     $(revokes access to segment for all processes
       forces access to be recomputed)
     EXCEPTIONS
       unmounted_volume(procuid, voluid);
       no_segment(procuid, voluid, seguid);
       write_not_allowed(procuid, h_seg_al(seguid));
     DELAY_UNTIL FORALL proc : h_proc_exists(proc) :
       (FORALL segno : h_kst_valid(proc, segno) AND
                       h_kst_seguid(proc, segno) = seguid :
         h_kst_seg_io_count(proc, segno) = 0);
     EFFECTS
       FORALL proc : h_proc_exists(proc) :
       (FORALL segno : h_kst_valid(proc, segno) AND
                       h_kst_seguid(proc, segno) = seguid :
         'h_kst_access_defined(proc, segno) = FALSE;

OFUN revoke_vol_access(voluid; procuid);
     $(revokes access to all segments on a volume)
     EFFECTS
       FORALL seguid : h_seg_exists(voluid, seguid) :
         EFFECTS_OF revoke_access(voluid, seguid, procuid);

OFUN init_address_space(procuid);
     $(initializes process address space)
     EFFECTS
       'h_proc_kst(procuid) = template_kst;
       'h_proc_activity_levels(procuid) = VECTOR (FOR i FROM 1
                                                  TO pl_size :
         IF i = init_level THEN TRUE ELSE FALSE);
       'h_level_handler_exists(procuid, init_level) = TRUE;
       'h_level_ring(procuid, init_level) = init_exec_pt.ring;
       'h_level_segno(procuid, init_level) = init_exec_pt.segno;
       'h_level_offset(procuid, init_level) = init_exec_pt.offset;

OFUN purge_address_space(procuid);
     $(releases all segments and devices in address space of process)
     EFFECTS
       FORALL segno : h_kst_valid(procuid, segno) :
         EFFECTS_OF release_segno(segno, procuid);
       FORALL devno : h_kdt_valid(procuid, devno) :
         EFFECTS_OF release_devno(devno, procuid);

OVFUN read_seg(segno; offset)[procuid] -> word;
     $(reads a word from a segment)
     DEFINITIONS
       ring_number IS h_proc_ring(procuid);
       page_number pageno IS (offset + page_size)/page_size;
```

```
      volume_uid voluid IS h_kst_voluid(procuid, segno);
      segment_uid seguid IS h_kst_seguid(procuid, segno);
    EXCEPTIONS
      invalid_segno(procuid, segno);
      unmounted_volume(procuid, voluid);
      no_segment(procuid, voluid, seguid);
      undefined_access(procuid, segno);
      no_read_permission(procuid, ring, segno);
      out_of_bounds(procuid, segno, offset);
    EFFECTS
      word = EFFECTS_OF read(seguid, offset);
      ~h_seg_paged(seguid) =>
           'h_kst_seg_used(procuid, segno) = TRUE;

OFUN write_seg(segno; offset; word)[procuid];
    $(writes a word of a segment)
    DEFINITIONS
      ring_number IS h_proc_ring(procuid);
      page_number pageno IS (offset + page_size)/page_size;
      volume_uid voluid IS h_kst_voluid(procuid, segno);
      segment_uid seguid IS h_kst_seguid(procuid, segno);
    EXCEPTIONS
      invalid_segno(procuid, segno);
      unmounted_volume(procuid, voluid);
      no_segment(procuid, voluid, seguid);
      undefined_access(procuid, segno);
      no_write_permission(procuid, ring, segno);
      out_of_bounds(procuid, segno, offset);
    EFFECTS
      EFFECTS_OF write(seguid, offset, word);
      ~h_seg_paged(seguid) =>
           'h_kst_seg_used(procuid, segno) = TRUE AND
           'h_kst_seg_modified(procuid, segno) = TRUE;

OFUN execute_seg(segno; offset)[procuid];
    $(executes a word from a segment)
    DEFINITIONS
      ring_number IS h_proc_ring(procuid);
      page_number pageno IS (offset + page_size)/page_size;
      volume_uid voluid IS h_kst_voluid(procuid, segno);
      segment_uid seguid IS h_kst_seguid(procuid, segno);
    EXCEPTIONS
      invalid_segno(procuid, segno);
      unmounted_volume(procuid, voluid);
      no_segment(procuid, voluid, seguid);
      undefined_access(procuid, segno);
      no_execute_permission(procuid, ring, segno);
      out_of_bounds(procuid, segno, offset);
    EFFECTS
      EFFECTS_OF allocate_page(seguid, pageno);
      'h_proc_segno(procuid) = segno;
      'h_proc_offset(procuid) = offset;
      IF h_seg_paged(seguid)
      THEN EFFECTS_OF set_page_used_indicator(seguid, pageno)
      ELSE 'h_kst_seg_used(procuid, segno) = TRUE;

OFUN call(segno; offset)[procuid];
    $(initiates inter-ring inward movement for process)
    DEFINITIONS
      ring_number IS h_proc_ring(procuid);
      volume_uid voluid IS h_kst_voluid(procuid, segno);
      segment_uid seguid IS h_kst_seguid(procuid, segno);
    EXCEPTIONS
      invalid_segno(procuid, segno);
      unmounted_volume(procuid, voluid);
      no_segment(procuid, voluid, seguid);
      undefined_access(procuid, segno);
      outside_call_bracket(procuid, ring, segno);
      invalid_call_limiter(offset);
    EFFECTS
      'h_proc_ring(procuid) = MIN({h_proc_ring(procuid),
                                   h_kst_rb(procuid, segno)[2]});
      'h_proc_segno(procuid) = segno;
      'h_proc_offset(procuid) = offset;

OFUN return(segno; offset; ring)[procuid];
    $(initiates inter-ring outward movement for process)
    EXCEPTIONS
      inward_return(procuid, ring);
    EFFECTS
      'h_proc_ring(procuid) = ring;
      'h_proc_segno(procuid) = segno;
      'h_proc_offset(procuid) = offset;

OFUN wire_seg(segno)[procuid];
    $(wires segment into main memory)
    DEFINITIONS
      volume_uid voluid IS h_kst_voluid(procuid, segno);
      segment_uid seguid IS h_kst_seguid(procuid, segno);
    EXCEPTIONS
      invalid_segno(procuid, segno);
      unmounted_volume(procuid, voluid);
      no_segment(procuid, voluid, seguid);
      undefined_access(procuid, segno);
      segment_wired(procuid, segno);
    EFFECTS
      'h_kst_seg_wired(procuid, segno) = TRUE;
      EFFECTS_OF change_wire_count(seguid, 1);

OFUN unwire_seg(segno)[procuid];
    $(unwires segment from main memory)
    DEFINITIONS
```

```
      volume_uid voluid IS h_kst_voluid(procuid, segno);
      segment_uid seguid IS h_kst_seguid(procuid, segno);
    EXCEPTIONS
      invalid_segno(procuid, segno);
      unmounted_volume(procuid, voluid);
      no_segment(procuid, voluid, seguid);
      undefined_access(procuid, segno);
      segment_not_wired(procuid, segno);
      segment_used_for_io(procuid, segno);
    EFFECTS
      'h_kst_seg_wired(procuid, segno) = FALSE;
      EFFECTS_OF change_wire_count(seguid, -1);

VFUN h_proc_exec_pt(procuid) -> exec_pt;
    $(returns execution point for process)
    HIDDEN;
    DERIVATION <h_proc_ring(procuid), h_proc_segno(procuid),
                h_proc_offset(procuid)>;

VFUN h_proc_ring(procuid) -> ring;
    $(returns effective ring of execution for process)
    HIDDEN;
    INITIALLY ring = ?;

VFUN h_proc_segno(procuid) -> segno;
    $(returns segment number of execution point for process)
    HIDDEN;
    INITIALLY segno = ?;

VFUN h_proc_offset(procuid) -> offset;
    $(returns segment offset of execution point for process)
    HIDDEN;
    INITIALLY offset = ?;

VFUN h_level_handler_exists(procuid; level) -> b;
    $(returns true if level handler is defined for process)
    HIDDEN;
    INITIALLY b = FALSE;

VFUN h_proc_activity_levels(procuid) -> activity_levels;
    $(returns activity level indicators for process)
    HIDDEN;
    INITIALLY activity_levels = ?;

VFUN h_proc_level(procuid) -> level;
    $(returns execution point level for process)
    HIDDEN;
    INITIALLY level = ?;

VFUN h_level_exec_pt(procuid; level) -> exec_pt;
    $(returns execution point for level handler for process)
    HIDDEN;
    DERIVATION <h_level_ring(procuid, level), h_level_segno(procuid,
               level), h_level_offset(procuid, level)>;

VFUN h_level_ring(procuid; level) -> ring;
    $(returns effective ring of level handler for process)
    HIDDEN;
    INITIALLY ring = ?;

VFUN h_level_segno(procuid; level) -> segno;
    $(returns segment number of level handler for process)
    HIDDEN;
    INITIALLY segno = ?;

VFUN h_level_offset(procuid; level) -> offset;
    $(returns segment offset of level handler for process)
    HIDDEN;
    INITIALLY offset = ?;

VFUN level_handler_exec_pt(level)[procuid] -> exec_pt;
    $(external form of h_level_exec_pt)
    EXCEPTIONS
      undefined_level(procuid, level);
    DERIVATION h_level_exec_pt(procuid, level);

VFUN proc_activity_levels()[procuid] -> activity_levels;
    $(external form of h_proc_activity_levels)
    DERIVATION h_proc_activity_levels(procuid);

VFUN h_kst_seg_wired(procuid; segno) -> b;
    $(returns true if segment wired by process)
    HIDDEN;
    INITIALLY b = FALSE;

VFUN h_kst_seg_used(procuid; segno) -> b;
    $(returns true if unpaged segment used by process)
    HIDDEN;
    INITIALLY b = FALSE;

VFUN h_kst_seg_modified(procuid; segno) -> b;
    $(returns true if unpaged segment modified by process)
    HIDDEN;
    INITIALLY b = FALSE;

OFUN decrement_seg_io_count(procuid; segno);
    $(decrements number of asynchronous io operations on segment)
    EFFECTS
      'h_kst_seg_io_count(procuid, segno) =
                    h_kst_seg_io_count(procuid, segno) - 1;

VFUN h_kst_seg_io_count(procuid; segno) -> io_count;
```

```
     $(returns number of asynchronous io operations on segment)
     HIDDEN;
     INITIALLY io_count = 0;

VFUN h_kst_valid(procuid; segno) -> b;
     $(returns true if segment number is valid for process)
     HIDDEN;
     INITIALLY b = FALSE;

VFUN h_kst_access_defined(procuid; segno) -> b;
     $(returns true if access defined to segment for process)
     HIDDEN;
     INITIALLY b = FALSE;

FUN h_kst_rb(procuid; segno) -> rb;
     $(returns ring brackets of segment for process)
     HIDDEN;
     INITIALLY rb = ?;

VFUN h_kst_mode(procuid; segno) -> mode;
     $(returns access mode to segment for process)
     HIDDEN;
     INITIALLY mode = ?;

VFUN h_kst_seguid(procuid; segno) -> seguid;
     $(returns seguid of assigned segno for process)
     HIDDEN;
     INITIALLY seguid = ?;

VFUN h_kst_voluid(procuid; segno) -> voluid;
     $(returns voluid of assigned segno for process)
     HIDDEN;
     INITIALLY voluid = ?;

VFUN kst_voluid(segno)[procuid] -> voluid;
     $(external form of h_kst_voluid)
     EXCEPTIONS
       invalid_segno(procuid, segno);
     DERIVATION h_kst_voluid(procuid, segno);

VFUN kst_seguid(segno)[procuid] -> seguid;
     $(external form of h_kst_seguid)
     EXCEPTIONS
       invalid_segno(procuid, segno);
     DERIVATION h_kst_seguid(procuid, segno);

VFUN kst_mode(segno)[procuid] -> mode;
     $(external form of h_kst_mode)
     EXCEPTIONS
       invalid_segno(procuid, segno);
       undefined_access(procuid, segno);
     DERIVATION h_kst_mode(procuid, segno);

VFUN kst_rb(segno)[procuid] -> rb;
     $(external form of h_kst_rb)
     EXCEPTIONS
       invalid_segno(procuid, segno);
       undefined_access(procuid, segno);
     DERIVATION h_kst_rb(procuid, segno);

OFUN assign_devno(devuid; devno)[procuid];
     $(assigns a device number to a device for process)
     EXCEPTIONS
       devno_in_use(procuid, devno);
       no_device(procuid, devuid);
       device_in_use(devuid);
     EFFECTS
       'h_kdt_valid(procuid, devno) = TRUE;
       'h_kdt_devuid(procuid, devno) = devuid;
       'h_kdt_access_defined(procuid, devno) = FALSE;
       'h_kdt_dev_used(procuid, devno) = FALSE;
       'h_kdt_dev_modified(procuid, devno) = FALSE;
       EFFECTS_OF assign_device(procuid, devuid, devno);

OFUN release_devno(devno)[procuid];
     $(releases a device number for a process)
     EXCEPTIONS
       invalid_devno(procuid, devno);
       device_active(procuid, devno);
     EFFECTS
       'h_kdt_valid(procuid, devno) = FALSE;
       EFFECTS_OF release_device(h_kdt_devuid(procuid, devno));

OVFUN give_device_access(devno; mode; rb; level)[procuid] -> given_mode;
     $(gives a process access to a device and returns given mode)
      DEFINITIONS
        device_uid devuid IS h_kdt_devuid(procuid, devno);
        access_level al IS h_dev_al(devuid);
        BOOLEAN trusted IS h_proc_trusted(procuid);
        BOOLEAN rw IS h_read_write_allowed(trusted, h_proc_al(procuid),
                                           al);
        access_mode max_mode IS VECTOR (rw, rw, trusted);
      EXCEPTIONS
        invalid_device_ring_brackets(rb);
        undefined_level(procuid, level);
        invalid_devno(procuid, devno);
        no_device(procuid, devuid);
      EFFECTS
        given_mode = VECTOR (FOR i FROM 1 TO 3 :
                             mode[i] AND max_mode[i]);
        'h_kdt_mode(procuid, devno) = given_mode;
        'h_kdt_rb(procuid, devno) = rb;
      'h_kdt_level(procuid, devno) = level;
      'h_kdt_access_defined(procuid, devno) = TRUE;

OFUN revoke_device_access(devuid)[procuid];
    $(revokes access to device
      forces access to be recomputed)
    EXCEPTIONS
      no_device(procuid, devuid);
      write_not_allowed(procuid, h_dev_al(devuid));
    EFFECTS
      h_dev_initiated(devuid) =>
        'h_kdt_access_defined(h_dev_initiator(devuid),
                              h_dev_devno(devuid)) = FALSE;

OVFUN sync_device_read(devno; opcode)[procuid] -> word;
    $(reads a word from a device synchronously)
    DEFINITIONS
      ring_number ring IS h_proc_ring(procuid);
      device_uid devuid IS h_kdt_devuid(procuid, devno);
    EXCEPTIONS
      invalid_devno(procuid, devno);
      no_device(procuid, devuid);
      undefined_device_access(procuid, devno);
      invalid_device_read(procuid, ring, devuid, devno, opcode);
    EFFECTS
      word = h_dev_contents(devuid);
      'h_kdt_dev_used(procuid, devno) = TRUE;

OFUN sync_device_write(devno; opcode; word)[procuid];
    $(writes a word to device synchronously)
    DEFINITIONS
      ring_number ring IS h_proc_ring(procuid);
      device_uid devuid IS h_kdt_devuid(procuid, devno);
    EXCEPTIONS
      invalid_devno(procuid, devno);
      no_device(procuid, devuid);
      undefined_device_access(procuid, devno);
      invalid_device_write(procuid, ring, devuid, devno, opcode);
    EFFECTS
      EFFECTS_OF write_device(devuid, word);
      'h_kdt_dev_used(procuid, devno) = TRUE;
      'h_kdt_dev_modified(procuid, devno) = TRUE;

OFUN connect_device_read(devno; segno; offset; range)[procuid];
    $(starts an asynchronous read operation from a device)
    DEFINITIONS
      ring_number ring IS h_proc_ring(procuid);
      device_uid devuid IS h_kdt_devuid(procuid, devno);
      segment_uid seguid IS h_kst_seguid(procuid, segno);
      volume_uid voluid IS h_kst_voluid(procuid, segno);
      page_number pageno IS (offset + page_size)/page_size;
    EXCEPTIONS
      invalid_devno(procuid, devno);
      no_device(procuid, devuid);
      undefined_device_access(procuid, devno);
      no_device_read_permission(procuid, ring, devno);
      device_active(procuid, devno);
      invalid_segno(procuid, segno);
      unmounted_volume(procuid, voluid);
      no_segment(procuid, seguid);
      undefined_access(procuid, segno);
      segment_not_wired(procuid, segno);
      max_io_operations(procuid, segno);
      invalid_memory_limits(procuid, devuid, seguid, offset, range);
      no_write_memory_permission(procuid, devuid, ring, segno);
    EFFECTS
      EFFECTS_OF set_device_active(devuid);
      'h_kdt_dev_used(procuid, devno) = TRUE;
      'h_kst_seg_io_count(procuid, segno) =
                     h_kst_seg_io_count(procuid, segno) + 1;
      IF h_dev_map_type(devuid) = mapped
      THEN 'h_kdt_segno(procuid, devno) = segno AND
           'h_kdt_ring(procuid, devno) = ring
      ELSE IF h_seg_paged(seguid)
           THEN EFFECTS_OF set_page_indicators(seguid, pageno)
           ELSE 'h_kst_seg_used(procuid, segno) = TRUE AND
                'h_kst_seg_modified(procuid, segno) = TRUE;

OFUN async_device_read(devuid; segno; offset);
    $(asynchronously read a word from mapped device to segment)
    DEFINITIONS
      process_uid procuid IS h_dev_initiator(devuid);
      device_number devno IS h_dev_devno(devuid);
      ring_number ring IS h_kdt_ring(procuid, devno);
      segment_uid seguid IS h_kst_seguid(procuid, segno);
    EXCEPTIONS
      device_not_active(devuid);
      invalid_segment_reference(procuid, devno, segno);
      no_write_permission(procuid, ring, segno);
      out_of_bounds(procuid, segno, offset);
    EFFECTS
      EFFECTS_OF write_seg(segno, offset, h_dev_contents(devuid),
                      procuid);
      ~h_seg_paged(seguid) =>
           'h_kst_seg_used(procuid, segno) = TRUE AND
           'h_kst_seg_modified(procuid, segno) = TRUE;

OFUN connect_device_write(devno; segno; offset; range)[procuid];
    $(starts an asynchronous write operation to a device)
    DEFINITIONS
      ring_number ring IS h_proc_ring(procuid);
      device_uid devuid IS h_kdt_devuid(procuid, devno);
```

```
        segment_uid seguid IS h_kst_seguid(procuid, segno);
        volume_uid voluid IS h_kst_voluid(procuid, segno);
        page_number pageno IS (offset + page_size)/page_size;
      EXCEPTIONS
        invalid_devno(procuid, devno);
        no_device(procuid, devuid);
        undefined_device_access(procuid, devno);
        no_device_write_permission(procuid, ring, devno);
        device_active(procuid, devno);
        invalid_segno(procuid, segno);
        unmounted_volume(procuid, voluid);
        no_segment(procuid, seguid);
        undefined_access(procuid, segno);
        segment_not_wired(procuid, segno);
        max_io_operations(procuid, segno);
        invalid_memory_limits(procuid, devuid, seguid, offset, range);
        no_read_memory_permission(procuid, devuid, ring, segno);
      EFFECTS
        EFFECTS_OF set_device_active(devuid);
        'h_kdt_dev_used(procuid, devno) = TRUE;
        'h_kdt_dev_modified(procuid, devno) = TRUE;
        'h_kst_seg_io_count(procuid, segno) =
                          h_kst_seg_io_count(procuid, segno) + 1;
        IF h_dev_map_type(devuid) = mapped
        THEN 'h_kdt_segno(procuid, devno) = segno AND
             'h_kdt_ring(procuid, devno) = ring
        ELSE IF h_seg_paged(seguid)
             THEN EFFECTS_OF set_page_used_indicator(seguid, pageno)
             ELSE 'h_kst_seg_used(procuid, segno) = TRUE;

  OFUN async_device_write(devuid; segno; offset);
     $(asynchronously write a word to mapped device from segment)
     DEFINITIONS
        process_uid procuid IS h_dev_initiator(devuid);
        device_number devno IS h_dev_devno(devuid);
        ring_number ring IS h_kdt_ring(procuid, devno);
        segment_uid seguid IS h_kst_seguid(procuid, segno);
     EXCEPTIONS
        device_not_active(devuid);
        invalid_segment_reference(procuid, devno, segno);
        no_read_permission(procuid, ring, segno);
        out_of_bounds(procuid, segno, offset);
     EFFECTS
        EFFECTS_OF write_device(devuid, EFFECTS_OF read_seg(segno, offset,
                                                            procuid));
        ~h_seg_paged(seguid) =>
             'h_kst_seg_used(procuid, segno) = TRUE;

  VFUN h_kdt_valid(procuid; devno) -> b;
     $(returns true if device number valid for process)
     HIDDEN;
     INITIALLY b = FALSE;

VFUN h_kdt_devuid(procuid; devno) -> devuid;
     $(returns devuid of assigned devno for process)
     HIDDEN;
     INITIALLY devuid = ?;

VFUN h_kdt_mode(procuid; devno) -> mode;
     $(returns access mode to device for process)
     HIDDEN;
     INITIALLY mode = ?;

VFUN h_kdt_rb(procuid; devno) -> rb;
     $(returns ring brackets of device for process)
     HIDDEN;
     INITIALLY rb = ?;

OFUN reset_device_used_indicator(devuid);
     $(resets local device used indicator)
     EFFECTS
       'h_kdt_dev_used(h_dev_initiator(devuid),
                       h_dev_devno(devuid)) = FALSE;

OFUN reset_device_modified_indicator(devuid);
     $(resets local device modified indicator)
     EFFECTS
       'h_kdt_dev_modified(h_dev_initiator(devuid),
                           h_dev_devno(devuid)) = FALSE;

VFUN h_kdt_dev_used(procuid; devno) -> b;
     $(returns true if device used by process)
     HIDDEN;
     INITIALLY b = FALSE;

VFUN h_kdt_dev_modified(procuid; devno) -> b;
     $(returns true if device modified by process)
     HIDDEN;
     INITIALLY b = FALSE;

VFUN h_kdt_level(procuid; devno) -> level;
     $(returns level of device handler for process)
     HIDDEN;
     INITIALLY level = ?;

VFUN h_kdt_segno(procuid; devno) -> segno;
     $(returns segment number being used by device)
     HIDDEN;
     INITIALLY segno =?;

VFUN h_kdt_ring(procuid; devno) -> eff_ring;
     $(returns effective ring for mediation of device asynchronous io)
     HIDDEN;
     INITIALLY eff_ring = ?;

VFUN kdt_devuid(devno)[procuid] -> devuid;
     $(external form of h_kdt_devuid)
     EXCEPTIONS
       invalid_devno(procuid, devno);
     DERIVATION h_kdt_devuid(procuid, devno);

VFUN kdt_mode(devno)[procuid] -> mode;
     $(external form of h_kdt_mode)
     EXCEPTIONS
       invalid_devno(procuid, devno);
       undefined_device_access(procuid, devno);
     DERIVATION h_kdt_mode(procuid, devno);

VFUN kdt_rb(devno)[procuid] -> rb;
     $(external form of h_kdt_rb)
     EXCEPTIONS
       invalid_devno(procuid, devno);
       undefined_device_access(procuid, devno);
     DERIVATION h_kdt_rb(procuid, devno);

VFUN kdt_level(devno)[procuid] -> level;
     $(external form of h_kdt_level)
     EXCEPTIONS
       invalid_devno(procuid, devno);
       undefined_device_access(procuid, devno);
     DERIVATION h_kdt_level(procuid, devno);

OFUN set_trap_handler(trapno; exec_pt; save_area)[procuid];
     $(establishes trap handler and save area for process)
     EXCEPTIONS
       inner_ring_handler(procuid, exec_pt.ring);
       existing_trap_handler(procuid, trapno, exec_pt.ring);
       invalid_segno(procuid, exec_pt.segno);
       invalid_segno(procuid, save_area.segno);
     EFFECTS
       'h_trap_handler_exists(procuid, trapno) = TRUE;
       'h_trap_ring(procuid, trapno) = exec_pt.ring;
       'h_trap_segno(procuid, trapno) = exec_pt.segno;
       'h_trap_offset(procuid, trapno) = exec_pt.offset;
       'h_trap_save_segno(procuid, trapno) = save_area.segno;
       'h_trap_save_offset(procuid, trapno) = save_area.offset;

OFUN trap(trapno; trap_info)[procuid];
     $(initiates trap handler for process)
     DEFINITIONS
       ring_number IS h_trap_ring(procuid, trapno);
       segment_number segno IS h_trap_save_segno(procuid, trapno);
       segment_offset offset IS h_trap_save_offset(procuid, trapno);
```

```
      volume_uid voluid IS h_kst_voluid(procuid, segno);
      segment_uid seguid IS h_kst_seguid(procuid, segno);
    EXCEPTIONS
      no_trap_handler(procuid, trapno);
      invalid_segno(procuid, segno);
      unmounted_volume(procuid, voluid);
      no_segment(procuid, voluid, seguid);
      undefined_access(procuid, segno);
      no_write_permission(procuid, ring, segno);
      out_of_bounds(procuid, segno, offset + LENGTH(trap_info));
    EFFECTS
      'h_trap_depth(procuid, trapno) = h_trap_depth(procuid, trapno) + 1;
      FORALL i : 1 <= i AND i <= LENGTH(trap_info) :
        EFFECTS_OF write_seg(segno, offset + i - 1, trap_info[i],
                     procuid);
      'h_trap_save_offset(procuid, trapno) = h_trap_save_offset(procuid,
                                     trapno) + LENGTH(trap_info);
      'h_proc_ring(procuid) = h_trap_ring(procuid, trapno);
      'h_proc_segno(procuid) = h_trap_segno(procuid, trapno);
      'h_proc_offset(procuid) = h_trap_offset(procuid, trapno);
      ~h_seg_paged(seguid) =>
           'h_kst_seg_used(procuid, segno) = TRUE AND
           'h_kst_seg_modified(procuid, segno) = TRUE;

OFUN return_from_trap(trapno; return_exec_pt; trap_info)[procuid];
    $(returns from trap handler for process)
    EXCEPTIONS
      no_trap_handler(procuid, trapno);
      no_trap_outstanding(procuid, trapno);
      inward_return(procuid, return_exec_pt.ring);
    EFFECTS
      'h_trap_depth(procuid, trapno) = h_trap_depth(procuid, trapno) - 1;
      'h_trap_save_offset(procuid, trapno) = h_trap_save_offset(procuid,
                                     trapno) - LENGTH(trap_info);
      'h_proc_ring(procuid) = return_exec_pt.ring;
      'h_proc_segno(procuid) = return_exec_pt.segno;
      'h_proc_offset(procuid) = return_exec_pt.offset;

VFUN h_trap_handler_exists(procuid; trapno) -> b;
    $(returns true if trap handler exists for process)
    HIDDEN;
    INITIALLY b = FALSE;

VFUN h_trap_depth(procuid; trapno) -> n;
    $(returns number of unprocessed traps for process)
    HIDDEN;
    INITIALLY n = 0;

VFUN h_trap_exec_pt(procuid; trapno) -> exec_pt;
    $(returns execution point of trap handler for process)
    HIDDEN;
```

```
     DERIVATION <h_trap_ring(procuid, trapno), h_trap_segno(procuid,
                 trapno), h_trap_offset(procuid, trapno)>;

VFUN h_trap_ring(procuid; trapno) -> ring;
     $(returns effective ring of trap handler for process)
     HIDDEN;
     INITIALLY ring = ?;

VFUN h_trap_segno(procuid; trapno) -> segno;
     $(returns segment number of trap handler for process)
     HIDDEN;
     INITIALLY segno = ?;

VFUN h_trap_offset(procuid; trapno) -> offset;
     $(returns segment offset of trap handler for process)
     HIDDEN;
     INITIALLY offset = ?;

VFUN h_trap_save_area(procuid; trapno) -> save_area;
     $(returns trap save area for process)
     HIDDEN;
     DERIVATION <h_trap_save_segno(procuid, trapno),
                 h_trap_save_offset(procuid, trapno)>;

VFUN h_trap_save_segno(procuid; trapno) -> segno;
     $(returns segment number of trap save area for process)
     HIDDEN;
     INITIALLY segno = ?;

VFUN h_trap_save_offset(procuid; trapno) -> offset;
     $(returns segment offset of trap save area for process)
     HIDDEN;
     INITIALLY offset = ?;

VFUN trap_handler_exec_pt(trapno)[procuid] -> exec_pt;
     $(external form of h_trap_exec_pt)
     EXCEPTIONS
       no_trap_handler(procuid, trapno);
     DERIVATION h_trap_exec_pt(procuid, trapno);

VFUN trap_save_area(trapno)[procuid] -> save_area;
     $(external form of h_trap_save_area)
     EXCEPTIONS
       no_trap_handler(procuid, trapno);
     DERIVATION h_trap_save_area(procuid, trapno);

OFUN set_level_handler(level; exec_pt)[procuid];
     $(establishes level handler for process)
     EXCEPTIONS
       inner_ring_handler(procuid, exec_pt.ring);
       existing_level_handler(procuid, level, exec_pt.ring);
```

```
      invalid_segno(procuid, exec_pt.segno);
    EFFECTS
      'h_level_handler_exists(procuid, level) = TRUE;
      'h_level_ring(procuid, level) = exec_pt.ring;
      'h_level_segno(procuid, level) = exec_pt.segno;
      'h_level_offset(procuid, level) = exec_pt.offset;

OFUN set_level(procuid; level; b);
    $(sets level activity flag for process)
    EFFECTS
      'h_proc_activity_levels(procuid) = VECTOR (FOR i FROM 1
                                                    TO pl_size :
        IF i = level THEN b ELSE h_proc_activity_levels(procuid)[i]);

OFUN dispatch_level(procuid);
    $(dispatches highest priority active level for process)
    DEFINITIONS
      INTEGER level IS MIN(i : h_proc_activity_levels(procuid)[i]);
    EFFECTS
      'h_proc_ring(procuid) = h_level_ring(procuid, level);
      'h_proc_segno(procuid) = h_level_segno(procuid, level);
      'h_proc_offset(procuid) = h_level_offset(procuid, level);
      'h_proc_level(procuid) = level;

OFUN change_level(procuid);
    $(changes activity level for process)
    DEFINITIONS
      INTEGER cur_level IS h_proc_level(procuid);
    EFFECTS
      'h_level_ring(procuid, cur_level) = h_proc_ring(procuid);
      'h_level_segno(procuid, cur_level) = h_proc_segno(procuid);
      'h_level_offset(procuid, cur_level) = h_proc_offset(procuid);
      EFFECTS_OF dispatch_level(procuid);

OFUN modify_activity_level(level; b)[procuid];
    $(modifies activity level indicator for process)
    EXCEPTIONS
      undefined_level(procuid, level);
    EFFECTS
      EFFECTS_OF set_level(procuid, level, b);

OFUN lev(level; b)[procuid];
    $(initiates level change for process)
    EXCEPTIONS
      undefined_level(procuid, level);
    EFFECTS
      EFFECTS_OF set_level(procuid, level, TRUE);
      b => EFFECTS_OF set_level(procuid, h_proc_level(procuid),
                                  FALSE);
      EFFECTS_OF change_level(procuid);
```

```
END_MODULE
```

```
MODULE  host_interfaces


     TYPES

level_number : {INTEGER ln : 0 <= ln AND ln <= max_ln};
category_set : {VECTOR_OF BOOLEAN cs : LENGTH(cs) = cs_size};
security_level : STRUCT (level_number sln;
                         category_set scs);
integrity_level : STRUCT (level_number iln;
                          category_set ics);
access_level : STRUCT (security_level sl;
                       integrity_level il);
process_uid : INTEGER;
volume_uid : INTEGER;
segment_uid : INTEGER;
ring_number : {INTEGER rn : 0 <= rn AND rn <= max_ring};
ring_brackets : {VECTOR_OF ring_number rb : LENGTH(rb) = 3};
segment_number : {INTEGER sn : 0 <= sn AND sn <= max_segno};
device_number : {INTEGER dn : 0 <= dn AND dn <= max_devno};
segment_offset : {INTEGER so : 0 <= so AND so <= max_offset};
access_mode : {VECTOR_OF BOOLEAN am : LENGTH(am) = 3};
segment_length : {INTEGER length : 1 <= length AND length <= max_offset
                                                              + 1};
machine_word : INTEGER;
device_control_data : VECTOR_OF machine_word;
device_status_data : VECTOR_OF machine_word;
message : INTEGER;
message_queue : VECTOR_OF message;


     DECLARATIONS

process_uid procuid;
volume_uid voluid;
segment_uid seguid;
ring_number ring;
ring_brackets rb;
segment_number segno;
device_number devno;
segment_offset offset;
access_mode mode;
segment_length length, range;
machine_word word;
device_control_data dev_control_data;
device_status_data dev_status_data;
message msg;
message_queue msg_queue;
access_level al;
BOOLEAN b;
INTEGER i;
```

PARAMETERS

```
INTEGER max_host_messages $(maximum size of host message queue);
device_control_data init_device_control_data $(initial control data);
```

DEFINITIONS

```
BOOLEAN unmounted_volume(procuid; voluid) IS
      IF ~h_vol_mounted(voluid) THEN TRUE
      ELSE ~h_read_allowed(h_proc_trusted(procuid), h_proc_al(procuid),
                           h_vol_min_al(voluid));

BOOLEAN no_segment(procuid; voluid; seguid) IS
      IF ~h_seg_exists(voluid, seguid) THEN TRUE
      ELSE ~h_read_allowed(h_proc_trusted(procuid), h_proc_al(procuid),
                           h_seg_visibility_al(seguid));

BOOLEAN invalid_devno(procuid; devno) IS
      ~h_kdt_valid(procuid, devno);

BOOLEAN invalid_segno(procuid; segno) IS
      ~h_kst_valid(procuid, segno);

BOOLEAN undefined_access(procuid; segno) IS
      ~h_kst_access_defined(procuid, segno);

BOOLEAN no_read_permission(procuid; ring; segno) IS
      ~(h_kst_mode(procuid, segno)[1] AND
         ring <= h_kst_rb(procuid, segno)[2]);

BOOLEAN no_write_permission(procuid; ring; segno) IS
      ~(h_kst_mode(procuid, segno)[2] AND
         ring <= h_kst_rb(procuid, segno)[1]);

BOOLEAN out_of_bounds(procuid; segno; offset) IS
      offset >= h_seg_length(h_kst_seguid(procuid, segno));

BOOLEAN message_pending(procuid; devno) IS
      h_host_msg_output_pending(procuid, devno);

BOOLEAN no_message_ready(procuid; devno) IS
      h_host_msg_input_pending(procuid, devno);

BOOLEAN max_messages(procuid; devno) IS
      LENGTH(h_host_msg_queue(procuid, devno)) = max_host_messages;

BOOLEAN interrupt_pending(procuid; devno) IS
      h_host_signal_pending(procuid, devno);
```

```
    EXTERNALREFS

FROM access_levels :
    level_number max_ln $(maximum level number);
    INTEGER cs_size $(category set size);
    VFUN h_read_allowed(b; al; al) -> b;
    VFUN h_write_allowed(b; al; al) -> b;

FROM processes :
    VFUN h_proc_al(procuid) -> al;
    VFUN h_proc_trusted(procuid) -> b;

FROM volumes :
    VFUN h_vol_min_al(voluid) -> al;
    VFUN h_vol_mounted(voluid) -> b;

FROM segments :
    segment_offset max_offset $(maximum segment offset);
    VFUN h_seg_exists(voluid; seguid) -> b;
    VFUN h_seg_visibility_al(seguid) -> al;
    VFUN h_seg_length(seguid) -> length;

FROM address_spaces :
    ring_number max_ring $(maximum ring number);
    segment_number max_segno $(maximum segment number);
    device_number max_devno $(maximum device number):
    VFUN h_kdt_valid(procuid; devno) -> b;
    VFUN h_kst_valid(procuid; segno) -> b;
    VFUN h_kst_access_defined(procuid; segno) -> b;
    VFUN h_kst_mode(procuid; segno) -> mode;
    VFUN h_kst_rb(procuid; segno) -> rb;
    OVFUN read_seg(segno; offset)[procuid] -> word;
    OFUN write_seg(segno; offset; word)[procuid];

    FUNCTIONS

OFUN send_message(segno; offset; range; devno)[procuid];
    $(sends data message associated with device devno to host process)
    DEFINITIONS
      ring_number IS h_proc_ring(procuid);
      volume_uid voluid IS h_kst_voluid(procuid, segno);
      segment_uid seguid IS h_kst_seguid(procuid, segno);
    EXCEPTIONS
      invalid_devno(procuid, devno);
      invalid_segno(procuid, segno);
      unmounted_volume(procuid, voluid);
      no_segment(procuid, voluid, seguid);
      undefined_access(procuid, segno);
```

```
        no_read_permission(procuid, ring, segno);
        out_of_bounds(procuid, segno, offset + range);
        message_pending(procuid, devno);
      EFFECTS
        'h_host_msg_output_pending(procuid, devno) = TRUE;
        'h_host_msg_output_segno(procuid, devno) = segno;
        'h_host_msg_output_offset(procuid, devno) = offset;
        'h_host_msg_output_length(procuid, devno) = range;

  OFUN receive_message(segno; offset; devno)[procuid];
      $(receives data message associated with device devno from host)
      DEFINITIONS
        ring_number IS h_proc_ring(procuid);
        volume_uid voluid IS h_kst_voluid(procuid, segno);
        segment_uid seguid IS h_kst_seguid(procuid, segno);
      EXCEPTIONS
        invalid_devno(procuid, devno);
        invalid_segno(procuid, segno);
        unmounted_volume(procuid, voluid);
        no_segment(procuid, voluid, seguid);
        undefined_access(procuid, segno);
        no_write_permission(procuid, ring, segno);
        out_of_bounds(procuid, segno, offset + h_host_msg_input_length(
                                                  procuid, devno));
        no_message_ready(procuid, devno);
      EFFECTS
        'h_host_msg_input_segno(procuid, devno) = segno;
        'h_host_msg_input_offset(procuid, devno) = offset;

  OFUN read_control_data(segno; offset; devno)[procuid];
      $(transfers device control data from host process)
      DEFINITIONS
        ring_number IS h_proc_ring(procuid);
        volume_uid voluid IS h_kst_voluid(procuid, segno);
        segment_uid seguid IS h_kst_seguid(procuid, segno);
      EXCEPTIONS
        invalid_devno(procuid, devno);
        invalid_segno(procuid, segno);
        unmounted_volume(procuid, voluid);
        no_segment(procuid, voluid, seguid);
        undefined_access(procuid, segno);
        no_write_permission(procuid, ring, segno);
        out_of_bounds(procuid, segno, offset + LENGTH(h_host_control_data(
                                                  procuid, devno)));
      EFFECTS
        FOR i FROM 1 TO LENGTH(h_host_control_data(procuid, devno)) :
        EFFECTS_OF write_seg(segno, offset - 1 + i,
                     h_host_control_data(procuid, devno)[i], procuid);

  OFUN write_status_data(segno; offset; range; devno)[procuid];
      $(transfers device status data to host process)
```

```
    DEFINITIONS
      ring_number IS h_proc_ring(procuid);
      volume_uid voluid IS h_kst_voluid(procuid, segno);
      segment_uid seguid IS h_kst_seguid(procuid, segno);
    EXCEPTIONS
      invalid_devno(procuid, devno);
      invalid_segno(procuid, segno);
      unmounted_volume(procuid, voluid);
      no_segment(procuid, voluid, seguid);
      undefined_access(procuid, segno);
      no_read_permission(procuid, ring, segno);
      out_of_bounds(procuid, segno, offset + range);
    EFFECTS
      'h_host_status_data(procuid, devno) = VECTOR (FOR i FROM 1
                                                    TO range :
        EFFECTS_OF read_seg(segno, offset - 1 + i, procuid));

OFUN send_wakeup(devno; msg)[procuid];
    $(sends wakeup message to host process associated with device devno)
    EXCEPTIONS
      invalid_devno(procuid, devno);
      max_messages(procuid, devno);
    DEFINITIONS
      INTEGER n IS LENGTH(h_host_msg_queue(procuid, devno));
    EFFECTS
      'h_host_msg_queue(procuid, devno) = VECTOR (FOR i FROM 1 TO n+1 :
        IF i <= n THEN h_host_msg_queue(procuid, devno)[i] ELSE msg);

OFUN send_host_signal(devno)[procuid];
    $(sends interrupt to host process associated with device devno)
    EXCEPTIONS
      invalid_devno(procuid, devno);
      interrupt_pending(procuid, devno);
    EFFECTS
      'h_host_signal_pending(procuid, devno) = TRUE;

VFUN h_host_msg_output_pending(procuid; devno) -> b;
    $(returns true if process has data message to host pending)
    HIDDEN;
    INITIALLY b = FALSE;

VFUN h_host_msg_output_segno(procuid; devno) -> segno;
    $(returns segment number of process data message to host)
    HIDDEN;
    INITIALLY segno = ?;

VFUN h_host_msg_output_offset(procuid; devno) -> offset;
    $(returns offset of process data message to host)
    HIDDEN;
    INITIALLY offset = ?;
```

```
VFUN h_host_msg_output_length(procuid; devno) -> length;
     $(returns length of process data message to host)
     HIDDEN;
     INITIALLY length = ?;

VFUN h_host_msg_input_pending(procuid; devno) -> b;
     $(returns true if process has data message from host pending)
     HIDDEN;
     INITIALLY b = FALSE;

VFUN h_host_msg_input_segno(procuid; devno) -> segno;
     $(returns segment number of process data message from host)
     HIDDEN;
     INITIALLY segno = ?;

VFUN h_host_msg_input_offset(procuid; devno) -> offset;
     $(returns offset of process data message from host)
     HIDDEN;
     INITIALLY offset = ?;

VFUN h_host_msg_input_length(procuid; devno) -> length;
     $(returns length of process data message from host)
     HIDDEN;
     INITIALLY length = ?;

VFUN h_host_control_data(procuid; devno) -> dev_control_data;
     $(returns device control data from host memory)
     HIDDEN;
     INITIALLY dev_control_data = init_device_control_data;

VFUN h_host_status_data(procuid; devno) -> dev_status_data;
     $(returns device status data from host memory)
     HIDDEN;
     INITIALLY dev_status_data = VECTOR ();

VFUN h_host_msg_queue(procuid; devno) -> msg_queue;
     $(returns contents of process message queue for host)
     HIDDEN;
     INITIALLY msg_queue = VECTOR ();

VFUN h_host_signal_pending(procuid; devno) -> b;
     $(returns true if signal pending for host)
     HIDDEN;
     INITIALLY b = FALSE;

VFUN host_output_message_pending(devno)[procuid] -> b;
     $(external form of h_host_msg_output_pending)
     EXCEPTIONS
       invalid_devno(procuid, devno);
     DERIVATION h_host_msg_output_pending(procuid, devno);
```

```
VFUN host_input_message_pending(devno)[procuid] -> b;
     $(external form of h_host_msg_input_pending)
     EXCEPTIONS
       invalid_devno(procuid, devno);
     DERIVATION h_host_msg_input_pending(procuid, devno);

VFUN host_input_message_length(devno)[procuid] -> length;
     $(external form of h_host_msg_input_length)
     EXCEPTIONS
       invalid_devno(procuid, devno);
       no_message_ready(procuid, devno);
     DERIVATION h_host_msg_input_length(procuid, devno);

END_MODULE
```

Appendix B

SPECIAL

REFERENCE MANUAL

by

Olivier Roubine

and

Lawrence Robinson

(2nd Edition)

Stanford Research Institute
Menlo Park, California

ABSTRACT

This document describes the specification language SPECIAL, which is a tool developed for the design of large software systems. The language is based on a methodology using the concept of a hierarchy of modules, and provides a convenient facility for the description of the properties of such modules. The syntax of the language is described, as well as the semantic notions related to its various constructs.

TABLE OF CONTENTS

1. INTRODUCTION

The design and proof of large software systems can be facilitated by their decomposition into modules, as suggested by Parnas (Par 72 a, b). A design methodology based on this idea has been developed at SRI (Neu 74; Rob 75 a, b), and applied to the construction of several large systems, including a secure operating system (Neu 75). A significant aspect of this work is the design of SPECIAL, a SPECIfication and Assertion Language, which is described here.

The reader is assumed to be already familiar with the methodology itself. The concepts underlying the construction of the language are not emphasized here and will be the subject of a forthcoming document.

This manual is chiefly concerned with the syntax of the language and the associated semantic rules. A general presentation of the language and the objects it manipulates is first given. The "paragraphs" that may appear in a module specification are then described. The part of the language concerned with algebraic and arithmetic expressions is described in Section 10. Further sections contain the precedence rules for binary operators and the rules for referencing an object (scope rules).

SPECIAL has evolved through several forms. The present edition of this manual corresponds to the second version of the language.

2. GENERAL PRESENTATION OF THE LANGUAGE

The macroscopic unit expressed in SPECIAL is the specification of a module or of a set of mapping-function expressions. The first word of a specification, MODULE (1) in the former case, or MAP in the latter, provides the necessary distinction.

We are chiefly concerned with module specifications, because specifications of mapping-function expressions use only a subset of the syntax available for module specifications.

---

(1) In the rest of this document, a word written in upper case refers to a SPECIAL reserved word.

A module specification consists of a sequence of paragraphs appearing between the header

```
MODULE <symbol>
```

and the keyword END_MODULE. The paragraphs are entitled TYPES, DECLARATIONS, PARAMETERS, DEFINITIONS, EXTERNALREFS and FUNCTIONS (or MAPPINGS, for mapping function specifications) and must appear in that order. A paragraph starts with a keyword (e.g., TYPES) and ends at the beginning of the next paragraph or at the end of the module. If a paragraph is empty, it must be omitted; i.e., two paragraph titles may not follow each other.

Thus, the general outline of a module is something like

```
MODULE <symbol> (2)
    DECLARATIONS
        .
        .
        .
    PARAMETERS
        .
        .
        .
    DEFINITIONS
        .
        .
        .
    EXTERNALREFS
        .
        .
        .
    FUNCTIONS
        .
        .
        .
 END_MODULE
```

The part of the language that deals with the objects of a module specification is known as the specification level; we describe later a more microscopic level known as the assertion level.

---

(2) Names appearing within angle brackets correspond to non-terminals of the grammar (see Appendix A); <symbol> is a particular non-terminal referring to any identifier that is not a reserved word; what is a legal identifier is described in Section 17.

3. THE OBJECTS USED IN THE LANGUAGE

The specification language manipulates several kinds of objects, all of which are "typed".

3.1. Types and Type Definitions

Each object manipulated by the language is associated with a type. For the descriptive purpose of this manual, it is sufficient to consider a type as a set of possible values.(3) An important aspect of the language is the description of the constraints imposed by types on arbitrary expressions. These constraints are the type rules of the language; they describe how operands of various types can be combined with particular operators to form an expression, and what type this expression will have.

The syntactic representation of a type is a type specification, which is a description of the properties of a type as a whole; type specifications are described in Section 5.3.

We distinguish various categories of types and describe them below:

Predefined types: The predefined types of the language are BOOLEAN, CHAR, INTEGER, and REAL.

Designator types: Designator types form a class of objects--designators--that are tokens for objects manipulated by the system being specified. Such objects may not be used as freely as, let us say, a number. All the objects with the same designator type are maintained by a particular module and can be created only by functions of this module (generally using the primitive NEW--see Section 11.15.).

Scalar types: Scalar types are types explicitly defined as sets of constants such that any two of them are disjoint (see the comments in Section 5.4.), e.g.,

```
direction : { left, right };
```

which defines "direction" as being a type, elements of which can have only one of the values "left" or "right"; those two symbols are implicitly declared as constants of the type "direction" by the type declaration. Note that such types are not ordered sets, and that only two operations are permitted on them: = and ~=. Objects of a

---

(3) The term possible values should be distinguished from meaningful values, which are the values for which a particular assertion is true. In particular, the range of a function is a subset of the type of its result.

scalar type also correspond to existing objects of the system but are not _maintained_ by a particular module.

Constructed types: The language contains two _unary_ type constructors, SET_OF and VECTOR_OF, used to define sets and vectors of objects of some given type, and two _n_-ary type constructors, ONE_OF and STRUCT, used to define _united types_ and _structured types_, respectively. Since we view a type as a set of possible values, ONE_OF corresponds to the union operator on sets, and STRUCT corresponds to the Cartesian product. Note that type constructors apply to types, not to elements: in particular, a set type defined as "SET_OF t" is a type that represents a set of sets; i.e., each element of this type is a set of values of type t.

Subtypes: The notion of subtype corresponds to that of subset: the language provides the possibility of defining a type as an explicit set of constants of some existing type or as the set of the elements in a particular type that satisfy some property. The particularity of these types is that, for the purpose of type checking, they define only objects of the principal type, since it would require at least a theorem prover to verify the closure of a subtype under all the operations defined on the principal type.

The term _primitive type_ refers to types that are sets of primitive values, one another disjoint; they are the predefined, designator, and scalar types. We can now define a _type_ as being a primitive type, a subtype, or the result of the application of some of the type-constructors to a certain number of types.

3.2. Objects

There is a distinction between the objects manipulated by a module and those manipulated by the specification language, the former being a subset of the latter. The objects of a module are functions and parameters (designator and scalar types, although being families of objects rather than objects, are sometimes referred to as objects of the module, principally because they can be referenced in other modules). In addition, the language deals with so-called definitions, formal and result arguments, and locally bound variables. Each object has a name, a syntactic class, a scope, and a type. The syntactic class is either simple or functional; a simple object can be referenced by a single identifier (its name), whereas a functional object is referenced by an identifier followed by a list of "actual parameters" which are expressions of the language. The scope of an object is the part of a module specification where an object can be referenced after having been declared (the concept of declaration is described in the following section); for instance, the objects of a module (functions and parameters) have the entire module specification as their scope.

3.2.1. Functions

A function is an object of the module itself. Its syntactic class is always functional, and, in addition, a function has to belong to one of three categories. A category is either VFUN, OFUN, or OVFUN, corresponding to the V-, O-, or OV-functions of the methodology. Functions are either described in the module itself or referred to as external by appearing in the EXTERNALREFS paragraph. A function has a (possibly empty) argument list; every time the function is referenced, it must be followed by a list of expressions, the elements of which are in one-to-one correspondence with the formal arguments, the type of the elements being the same for each pair. In the case of a V- or OV-function, the function also has a "result argument" of a certain type. We sometimes mention the type of a function, thereby meaning the type of its result argument.

3.2.2. Parameters

The parameters of a module are objects that receive a value at module initialization and cannot change thereafter. They can be either simple or functional objects, according to whether the value they receive is a simple value or a function (in the mathematical sense). They represent objects that are fixed in one particular module instantiation, but can be different in different instantiations. As already mentioned, parameters are objects of a module, and, as such, their scope is the entire module specification.

3.2.3. Definitions

A definition is a named object that can be textually replaced by a particular expression; it corresponds to what is generally known as a macro. Definitions can be either simple or functional, and they can be declared either in the DEFINITIONS paragraph, in which case their scope is the whole module specification, or in the DEFINITIONS section of a function, in the case of local definitions whose scope is the function specification.

Definitions differ from the general concept of macros in that the expression that is the body of the definition must be a valid expression when it is declared and not just when it is used (the rules pertaining to the use of definitions guarantee the latter condition if the former one is satisfied). This precludes the use of free variables inside definitions. One of the reasons for this handling of definitions is that it generally prevents the same name from standing for two totally different meanings depending on where it is used.

3.2.4. Formal Arguments

These are the variables appearing in the argument list of a function specification. They may be used freely within the function specification. Formal arguments are always simple objects of the language.

3.2.5. Result Arguments

A V or OV-function __must__ have one and only one result argument, a simple object that may be used in the function specification to refer to the value returned by that function.

3.2.6. Locally Bound Variables

Locally bound variables are objects whose binding remains in effect only within particular expressions, such as __quantified expressions__ (see FORALL and EXISTS--Section 11.10.) or set and vector constructors (see Sections 11.7. and 11.8.). Another important use of such variables is in the so-called LET-expressions (see Section 11.13).

4. BINDING AND THE CONCEPT OF DECLARATION

An object of the language is a rather abstract entity: to be manipulated with a minimal degree of convenience, an object must be associated with a name. Note that it does not make any difference, as far as the properties of a function are concerned, whether its first argument is named "a" or "z", provided that the chosen name is used consistently. In addition, an object must have a type and always has a "scope". We are thus confronted with the problem of associating a name with a particular object (e.g., argument), and also of associating an object with a particular type. What one actually does is to associate a name with a type, and a name with an object, which thus receives the type associated with the name.

4.1. Simple and Multiple Declarations

The association of a name with a type is performed by a particular construct of the language known as a __declaration__. A simple declaration has the form:

```
<simple declaration> ::= <type specification> <symbol>
```

(the syntax for a type specification is described in the next section). An extrapolation of this construct, referred to as a multiple declaration, can be used to associate several names with the same type specification, using the syntax:

```
<multiple declaration> ::= <simple declaration>
                         | <multiple declaration> ',' <symbol>
```

Generally, the association between a name and a particular object (the "binding") is performed implicitly by the place where the declaration of the name appears; for example:

```
        VFUN f1( INTEGER i ) -> BOOLEAN b;
```

means that:

* The name i is associated with the type INTEGER, and is bound to be the first (formal) argument of f1.
* Similarly, the name b is associated with the type BOOLEAN and designates the result argument of f1.
* Lastly, f1 itself is declared as a V-function of the type (INTEGER -> BOOLEAN).

4.2. Global Declarations and Their Use

It is often the case in specifications that declarations are an important portion of the whole specification and that type specifications themselves can be quite complicated. With this in mind, SPECIAL provides the possibility of separating declarations and bindings (sometimes called the "deferred binding" facility), in the case of formal and result arguments and locally bound variables. This separation is achieved by declaring a name (i.e., associating it with a type) in a global DECLARATIONS paragraph (see Section 6), and performing the binding by letting the name alone appear where a declaration was expected. The rules for using deferred binding are described below:

When a declaration is expected, two cases are possible. If there is indeed a declaration, the name is associated with the type appearing in the declaration and is bound to the relevant object, as determined by the lexical position of the declaration; if the name has already appeared in a global declaration, the new local declaration supersedes the global one for the scope of the object. On the other hand, if a single name appears instead of the declaration, the name designates the corresponding object, the latter being associated with the type appearing in the global declaration of the name. It is obviously an error to use a single name instead of a declaration if the name has not appeared in the DECLARATIONS paragraph.

Remark: In the case of formal arguments to definitions and functions, the language allows a list of multiple declarations separated by ";". However, if globally declared names are used instead of declarations, each name must be followed by a ";", and not by a ","; in other words, an argument list of the form

```
        ( INTEGER i, j; BOOLEAN b )
```

should be written, if i, j, and b had been declared globally

```
        ( i; j; b )
```

5. TYPES

The TYPES paragraph has two purposes: it allows the user to introduce the <u>designators</u> of the module and to associate names with type specifications (thus providing a macro-like facility for types). The names defined in the TYPES paragraph as particular type specifications are referred to as "named types".

5.1. <u>Designators</u>

A user-defined type, or designator, corresponds to a class of objects that are tokens for abstract objects implemented in the software system being specified. It must have a name, introduced by a declaration of the form:

```
<symbol> : DESIGNATOR;
```

The name introduced in this declaration will thereafter be a primitive type of its own.

5.2. <u>Named Types</u>

A type, as defined in Section 3.1., may involve an arbitrary number of type constructors, leading to a complex type specification. Such a type expression can be named in a statement like

```
<type name> : <type specification>;
```

<type name> is a new identifier that can textually replace the type specification appearing on the right-hand side of the equal sign.

5.3. <u>Type Specification</u>

A type specification is the syntactic representation of a type.

* A predefined type, a designator or a named type is represented by a symbol.

* A scalar type has a specification of the form:

```
<scalar type> ::= "{" <symbol> {"," <symbol>}* "}"   (4)
```

* A subtype has the syntax of a set-expression (see Section 11.7)

* A structured type is defined as:

---

(4) The syntax descriptions used in this manual follow the extended BNF conventions described in Appendix A

```
<structured type> ::= STRUCT '(' {<declaration> ';'}+ ')'
```

where <declaration> is

```
<declaration> ::= <simple declaration>
                | <multiple declaration>
```

Each declaration introduces a name that is associated with a particular component of a structure, a "field".

* A united type specification is:

```
<united type> ::= ONE_OF '(' <type specification>
                      {',' <type specification>}* ')'
```

A united type is a peculiar concept: We defined have earlier (Section 3.1.) a primitive type as a set of simple values disjoint from all the other primitive types (in particular, for a purist, the predefined types should include INTEGER and "NON_INTEGER_REAL", with REAL being defined as ONE_OF(INTEGER,NON_INTEGER_REAL)). A united type which is the union of some primitive types is also a set of simple values but is not necessarily disjoint from the primitive types. In addition, the operations defined on the component types are not defined on the united type, with the exception of '=' and '~=' which are defined on any type. The use of objects of united types must abide by certain rules, described later in this document (see in particular TYPECASE--Section 11.12, and coercion rules--Section 14).

* Sets and Vectors: These are constructed types specified by statements of the form:

```
<constructed type> ::= {SET_OF | VECTOR_OF} <type specification>
```

The TYPES paragraph contains an arbitrary number of type declarations and might look like:

```
    TYPES

capability, unique_identifier : DESIGNATOR;
machine_word : ONE_OF(INTEGER, capability);
access_rights : VECTOR_OF BOOLEAN;
```

the syntax being:

```
<types paragraph> ::= TYPES {<type declaration> ';'}+

<type declaration> ::= <symbol> { ',' <symbol>}*
                           ':' <type specification>
```

5.4. Additional Comments

The following remarks concern the use of scalar types and subtypes; those may not appear explicitly except when used to define a named type, which means that any type specification having the syntax of a set expression may appear only inside the TYPES paragraph or the EXTERNALREFS paragraph, and within these paragraphs, may not be embedded in another type specification. In addition, some special rules apply to the use of scalar types, since some problems might arise in cases like:

```
    TYPES

traffic_light : { green , yellow , red };
color : { red , yellow , green , cyan , blue , magenta };
```

Here, we have first to consider "green", "yellow", and "red" as three constants of type "traffic_light". Then, when seeing the second declaration, we must realize that "green", "yellow", and "red" are in fact constants of the type "color" and that "traffic_light" is a subtype of "color".

Thus, the definition of a scalar type is the following: A scalar type specification is a set of literal constants such that either none of them is ever used in any other type specification in the text, or some of them are used to define a subtype of the scalar type, i.e., they appear in a subset of the set_expression defining the scalar type.

This definition precludes, in particular, the following pathological case: Assume that "color" is defined as above, but the user wishes to introduce a more sophisticated traffic light, to cope with a particularly dangerous intersection, as:

```
traffic_light: { green , yellow , red , left_arrow }
```

thus creating the problem of defining the type of each of the constants, since "traffic_light" can no longer be considered a subtype of "color". Is "green" of type "color" or of type "traffic_light", or is it "ONE_OF(color, traffic_light)"? Rather than defining complicated rules to solve this problem, we prefer to forbid the intermixing of constants of different types (the problem raised here can be solved by defining a scalar type that is a superset of both "color" and "traffic_light").

Some further thought should also be given to the concept of structured types. One has to realize that these types allow the manipulation of tuples of heterogeneous objects, but that their purpose is not to define complex data structures such as trees, as is ordinarily done in modern programming languages (e.g., Pascal). It is in fact one of the goals of the methodology, of which SPECIAL is only a part, to allow the modules themselves to support the specification of complex data structures, and there is no reason for those to be provided directly by the language. Therefore, recursively defined structured types are not allowed.

6. DECLARATIONS

The DECLARATIONS paragraph contains declarations (i.e., name/type associations) for names the binding of which is deferred, as described in Section 4. If the user does not wish to use the deferred binding facility, the DECLARATIONS paragraph can be omitted.

A declaration associates a name with a type and stays in effect within the entire module.

The syntax for the DECLARATIONS paragraph is:

```
<declarations> ::= DECLARATIONS {<declaration> ";"}+
```

with

```
<declaration> ::= <type specification>
                        <symbol> {"," <symbol>}*
```

Example:

```
    DECLARATIONS
BOOLEAN  b;
INTEGER   i,j;
capability   c, c1;
SET_OF capability  sc;
```

7. PARAMETERS

Starting with the keyword PARAMETERS, this paragraph contains declarations for all the module parameters (see Section 3.2.2). Since parameters of the module can optionally have arguments, the syntax for parameter declarations is similar to the syntax for global declarations, with the option of appending a formal argument list after each symbol:

```
<parameter declaration> ::= <type specification> <symbol>
                            [<formalargs>]
                            {',' <symbol>[<formalargs>]}* ';'

<formalargs> ::= '(' [<declaration> {';' <declaration>}*] ')'
```

8. DEFINITIONS

Definitions are syntactic shorthands for arbitrary expressions. They are defined in the DEFINITIONS paragraph; that is, the name of a definition is associated with the expression for which it stands by a declaration of the form:

```
<definition> ::= <typespecification> <symbol> [<formalargs>]
                     IS <expression> ';'
```

The type specification must correspond to the type of the expression, this redundancy being considered beneficial to the clarity of specifications. The expression appearing in the definition may reference only the formal arguments, if any, and the objects that are bound globally, i.e., parameters, global definitions and functions.

The list of formal arguments may be omitted, or may be present but empty; there is a difference in that a definition with an empty argument list must be referenced as a functional expression throughout the specification (e.g., "foo()"), whereas a definition with no argument list must be referenced as a simple term (e.g., "fie"). There is no semantic difference between these two cases, the only difference being purely syntactic.

9. EXTERNALREFS

Although the decomposition of a software system into modules is intended to abolish the intermodule assumptions at the specification level, this is not always possible. A mechanism for referring to the functions, parameters, and designator or scalar types of other modules is therefore needed in order to describe these intermodule assumptions. This is the purpose of the EXTERNALREFS paragraph.

The paragraph itself is divided into as many groups as there are references to different modules. Each group starts with the header:

```
FROM <symbol> :
```

where <symbol> is the name of an external module, and the group contains a list of declarations and/or function headers.

A declaration has the regular declaration syntax, as in Section 6, or the syntax of a designator or scalar type declaration, as described in Section 5.

A function header is

```
{ VFUN | OFUN | OVFUN } <symbol> <formalargs>
                    [ '[' <declaration> {';' <declaration>}* ']' ]
                    [ '->' <declaration> ] ';'
```

If the first term is VFUN or OVFUN, then the result part (i.e., '->' <declaration>) should be present, and not for OFUN.

As will be shown, this syntax is also used to begin internal function specifications.

Example:

```
    EXTERNALREFS

FROM capabilities:
capability, unique_identifier: DESIGNATOR;
access_type: { read , write , execute , modify , append };
OVFUN  create_capability() -> c;
OVFUN  restrict_acces(capability c1,
                      VECTOR_OF access_type atv)
                        -> capability c;
 VECTOR_OF access_type atv;
VFUN  get_uid(c) -> unique_identifier u;

FROM segments:
OVFUN  create_segment(INTEGER i,
                      VECTOR_OF capability siv)
                     -> capability s;
VFUN  h_seg_exists(unique_identifier u) -> BOOLEAN b;
INTEGER  maxsegs;
```

10. FUNCTIONS

The FUNCTIONS paragraph contains the specifications for all the functions of the module and is therefore its most important part.

The methodology deals with OFUN, OVFUN, and VFUN which all have their particularities, although also some common points.

A function definition consists of a header, possibly a set of local macros in a DEFINITIONS subsection, and other subsections which will be characterized for each class of function.

10.1. The Function Header

Similar to that described in Section 9, it looks like

```
{ VFUN | OFUN | OVFUN } <symbol> <formalargs>
              [ '[' <declaration> {';' <declaration>}* ']' ]
              [ '->' <declaration> ] ';'
```

As in definitions or external references, any of the formal arguments, as well as the result argument, may have been declared in the DECLARATIONS paragraph and thus not be preceded by a type specification.

10.2. Implicit Arguments

The optional list of declarations enclosed within square brackets in the function header is the list of "implicit arguments". An implicit argument to a function is an argument that should be provided by the system, and not by the user, when the function is called (in the actual implementation). In terms of specifications, any reference to a function that has been defined with implicit arguments should contain as many actual arguments as the sum of the number of formal <u>and</u> implicit arguments appearing in the function header.

Example:

```
A function whose header is:
    VFUN f(INTEGER i)[process_id p] -> INTEGER ];
may be referenced as a function of two arguments, e.g.,
    f(0, p1)
```

10.3. Local Definitions

By introducing a section with exactly the same syntax as the DEFINITIONS paragraph, the user has the possibility of defining macros that are known only inside one function specification. Such definitions have the advantage of being able to reference the arguments of the function that have only a restricted binding.

10.4. The Subsections of a V-function

The two remaining sections of a V-function describe its formal properties. A V-function may be visible or hidden, and--independently--derived or primitive.

If a V-function is visible, it may be referenced in programs outside the module, so a list of exception conditions must appear in the function specification. This EXCEPTIONS subsection is similar to the one appearing in O- and OV-function specifications and is detailed in Section 10.5.

On the other hand, a hidden V-function may not be called outside the module and needs no EXCEPTIONS subsection. In such a case, the reserved word HIDDEN appears in its place.

The last subsection of a V-function specification can either define the initial value of the V-function (for primitive V-functions) or explicitly describe the functions value in terms of other objects of the module (for derived V-functions). The former case is described through an assertion of the form

```
        INITIALLY <expression> ';'
```

where <expression> is an arbitrary Boolean expression. In the latter case, the syntax is

```
        DERIVATION <expression> ';'
```

Here, the expression must have the type of the result argument of the function.

comments:

* The set of all primitive V-functions entirely defines the current state of the machine described by the module. The set of their initial values thus defines the initial state of the module.

* Derived V-functions are often used to restrict or to package the information visible outside the module.

* Note that a V-function that is both hidden and derived is very similar to a global definition.

The general syntax for a V-function specification is thus

```
VFUN <symbol> ([<declaration> {',' <declaration>}* ])
                              -> <declaration> ';'
[DEFINITIONS { <definition> ';' }+ ]
{HIDDEN ';' | [ EXCEPTIONS { <expression> ';'}+ ]}
{INITIALLY | DERIVATION} <expression> ';'
```

10.5. The Subsections of O- and OV-functions

As opposed to V-functions, O- and OV-functions are always visible(5) and consequently may have an EXCEPTIONS subsection. Since both perform some operations that affect the state of the module, they also have an EFFECTS subsection.

10.5.1. The EXCEPTIONS Subsection

This subsection contains a list of conditions that, if any is satisfied (i.e., evaluates to TRUE when the function is called), should prevent any effect from taking place and/or any value from being returned. If an exception condition is raised, control is returned to the calling program with "appropriate" notification to the caller. Exactly how this notification is made depends on the conventions of the programming language; we mention here only the constraints imposed by the specification language.

An exception condition is a Boolean expression; the order in which the conditions are listed is important: The meaning of the EXCEPTIONS subsection is that each condition should be checked <u>in turn</u>, and only if all the conditions are false should the effects take place and/or the value be returned. The calling program may depend on the order in which the exception conditions are checked. For example, a reasonable EXCEPTIONS subsection might be:

```
      EXCEPTIONS
  h_interrupt_set(i) = FALSE;
  h_int_handler_offset(i) = ?;
```

(where the two functional expressions are likely to be V-function references), which might mean that i does not refer to any interrupt in the first exception and that, if it indeed does, no interrupt handler exists for it in the second exception; this illustrates that the second exception might be meaningless had not the first one been checked before.

There is also a particular construct intended to be applied only in the EXCEPTIONS subsection:

```
EXCEPTIONS_OF <call> ';'
```

where <call> should be a reference to an external function. The

---

(5) Probably the only justification for this fact is that the need for hidden O or OV-functions has not yet been felt.

interpretation of this construct is that for each exception condition of the external function, there exists a logically equivalent exception condition in the current function. The order in which the exceptions appear in the external function is preserved, and the corresponding conditions are to be checked after all those that precede the "EXCEPTIONS_OF" expression.

The syntax for the EXCEPTIONS subsection can be described as

```
<exceptions> ::= EXCEPTIONS { <expression> ";"
                            | EXCEPTIONS_OF <call> ";" }+
```

Note that this expression forces at least one expression to occur in the EXCEPTIONS subsection; if the function has no exception, the whole subsection should be omitted.

10.5.2. The EFFECTS Subsection

The EFFECTS subsection contains a list of assertions that are Boolean expressions and have the following meaning: If the function is referenced and no exception is detected, then, after the function call, the conjunction of all the assertions appearing in the EFFECTS subsection may be assumed to be TRUE.

If the EFFECTS subsection of function f contains the assertions a1, a2,...,an, i.e., looks like

```
EFFECTS  a1; a2;...; an;
```

and the EXCEPTIONS subsection refers to e1,...,em (i.e., EXCEPTIONS e1; e2;...;em;) then, using Hoare's notation, a reference to the function f has the meaning

```
NOT(e1 OR e2 OR...OR em) {f} a1 AND a2 AND...AND an
```

Note that, in the EFFECTS subsection, the order of the assertions is irrelevant.

In the case of an OV-function, one of the assertions might state the equality of the result argument with some expression, thus defining the value returned by the function, but this is not mandatory; the result argument should, however, appear in at least one of the effects, in order not to be undefined.

10.5.3. The DELAY Conditions

In addition to the EXCEPTIONS and EFFECTS subsections, an O- or OV-function may contain an arbitrary number (although generally only one) of so-called DELAY expressions which have the syntax:

```
DELAY UNTIL <expression> ";"
```

where <expression> must be Boolean, and the conjunction of all the delay conditions can be taken as an input assertion to the function; in other terms, this means that the affects will not take place until all the conditions evaluate to TRUE.

This concludes our description of the macroscopic or specification level of the language. The microscopic (or assertion) level deals with the kind of construct we have been so far referring to as <expression> or assertion, with the distinction that an assertion is an expression of type BOOLEAN. This part of the language is described next.

11. EXPRESSIONS

The assertion language has a relatively large set of syntax rules for writing concise mathematical expressions to describe conditions related to a function behavior. In certain cases, the semantics associated with these rules may require a detailed description. This section explores the different constructs used in mathematical expressions; an interesting aspect to be considered is the set of type rules associated with each operator.

11.1. Atomic Expressions

These are the simplest kind of expression (from the syntactic standpoint); an atomic expression can be one of the following:

* Numeric constant: A numeric constant may be either of type INTEGER (any nonempty sequence of digits), or of type REAL (a string of digits followed by a period followed by another string of digits followed by the letter E followed by an optionally signed string of digits, where either the integer part or the fractional part may be omitted, and the decimal point or the exponent may also be omitted--but not both).

* Character and string constant: A constant of type CHAR is a single, printable ASCII character enclosed within two ` (back apostrophe). A string constant has the type "VECTOR_OF CHAR" and consists of any sequence of printable ASCII characters enclosed within two " (double-quote). The precise rules concerning character and string constants may depend on the implementation (see Section 17).

* Boolean constant: These are the keywords TRUE and FALSE.

* Symbolic constant: Any identifier appearing in a scalar type

declaration will subsequently be considered a constant of that type. In addition, the two symbols UNDEFINED and ? are two synonyms which, for each type, refer to a particular value of that type, different from any other value (the particular type is determined by the context where they appear). It is a part of the verification task to check that the implementation of any visible V- or OV-function can never return the value UNDEFINED.

* Reference to an object: The name of any object that has a binding and whose syntactic class is not functional is an atomic expression whose type is that of the object. Such references can be to parameters or definitions when they have been declared without arguments, formal arguments of functions and definitions, result arguments of V- and OV-functions, and simple variables within expressions where they have a binding (i.e., within set and vector constructors, quantified expressions, and expressions starting with LET or SOME).

11.2. V-function References

A reference to a V-function has the form

```
['] <symbol> <actual list>
```

where <actual list> is defined as

```
'(' [ <expression> {',' <expression>}* ] ')'
```

and is used as such in the rest of this document. It corresponds to the actual arguments, which must match in number and type the formal arguments of the function definition.

<symbol> should be the name of a V-function either defined in the module or appearing in the EXTERNALREFS paragraph.

The first symbol, ' , is optional. Its meaning, within the EFFECTS part of a function specification, is that the reference is made to the value of the V-function immediately _after_ all the effects of the O- or OV-function being specified have occurred; if it is omitted, the reference is to the value immediately _before_ the call. Note that this is meaningful only in the EFFECTS section, since in any other place there is no distinction between _before_ and _after_, and in the EXCEPTIONS subsection, a reference is always made _before_ any effect takes place.(6)

---

(6) We must add at this point an important historic note: Until now, all the module specifications derived from Parnas' ideas used quoted expressions to refer to _old_ values of V-functions. However, this notation could not be used consistently because in the same function specification, unquoted references would have two different meanings, depending whether they appeared in the exceptions or in the effects of the function. Permuting the convention takes care of this possible confusion.

The type of a V-function reference is the type of the function itself.

11.3. References to Functional Objects

We deal here with parameters and definitions introduced with an argument list. A reference to such objects has a syntax very similar to that of a V-function reference:

```
<symbol> <actual list>
```

with, for definitions, the usual meaning of a macro expansion, the type being that of the macro, and, in the case of a parameter, that of a mere reference to its value.

11.4. O- and OV-function References

A reference to an O- or OV-function is made by the use of the keyword EFFECTS_OF

```
EFFECTS-OF <symbol> <actual list>
```

In the case of O-functions, this expression is of type Boolean; its value is UNDEFINED if some exception is detected; it is FALSE if the effects of the O-function that is referenced are never satisfiable. A value of TRUE, on the contrary, means that all the assertions specified in the EFFECTS section of the corresponding O-function hold.

In the case of an OV-function, the type of the expression is that of the result argument of the function, and the expression itself is both a reference to the value of the OV-function and an indication that all its effects are true at the point where the EFFECTS_OF appears.

The next sections describe the operators used in the language. They are classified by the types on which they operate; the word "mode" appearing in a type description refers to a particular type which does not have to be specified. The discussion concerning precedence of various operators is, however, deferred to Section 12.

11.5. Boolean-Valued Operators

11.5.1. Binary Operators

AND and OR
Syntax: <expression> {AND | OR} <expression>
Type: BOOLEAN X BOOLEAN --> BOOLEAN
Meaning: usual conjunction or disjunction; *error* if one

of the operands is not Boolean.(7)

=>

Syntax: expression_1 => expression_2
Type: BOOLEAN X BOOLEAN --> BOOLEAN
Meaning: usual implication, i.e.,
(expression_2 OR NOT expression_1) = TRUE
error if one of the expressions is not Boolean.

11.5.2. Relational Operators

=, ~=

Syntax: exp_1 {= | ~=} exp_2
Type: mode X mode --> BOOLEAN
the types of exp_1 and exp_2 may be either:
any type, provided it is the same for both,(8)
or: any type for one expression and UNDEFINED for the other,
or: UNDEFINED for both;
Meaning: usual = or ≠

<, <=, >=, >

Syntax: exp_1 { > | >= | <= | < } exp_2
Type: number X number --> BOOLEAN
(where number is either INTEGER or REAL)
Meaning; the usual order relations on numbers.

11.6. Operations on Numbers

The operations on numbers are the usual +, -, * and /, where - can be either unary or binary; in addition, the operator MOD is also provided. The meaning and use do not depart from those generally encountered. Numbers are either of type INTEGER or of type REAL. Both types can be intermixed according to the following rules: A binary operation involving two integers will have the type INTEGER, and an operation where at least one of the operands is real will have the type REAL. In particular, the operator /, when applied to two integers, will yield the integer part of the quotient of the two operands, whereas if one of the operands is of type REAL, the result should be the quotient itself. In addition, some primitives are provided in the language for specifying the integer and fractional parts of a real (see Section 11.16.). The use of MOD is restricted to INTEGER operands.

All the binary operators on numbers, as well as those on Booleans, or on sets are left-associative.

---

(7) When the term *error* is used, it does not refer to any particular value, but rather means that under the conditions associated with it a specification would be incorrect, as far as SPECIAL is concerned. In other terms, it is considered as erroneous to say "1 AND TRUE" as to say "DO I = 1, 10" in a specification.

(8) For more about type matching rules, see Section 14.

11.7. Operations on Sets

11.7.1. Unary Operator

CARDINALITY

Syntax: CARDINALITY(<set expression>)
Type: set-mode --> INTEGER
Value: The number of elements in the set designated by its argument

11.7.2. Binary Operators

UNION

Syntax: <set expression> UNION <set expression>
Type: SET_OF m1 X SET_OF m2 --> SET_OF m3
where m3 is the larger of m1 and m2 if they match, and their union otherwise; note that, when the types of the two arguments do not match, the result will have the type of the minimal union of the two types, e.g.,

```
If the type of s1 is "SET_OF INTEGER"
and the type of s2 is "ONE_OF( SET_OF BOOLEAN, SET_OF CHAR)"
then s1 UNION s2 will be a
"ONE_OF( SET_OF ONE_OF( BOOLEAN, INTEGER),
                 SET_OF ONE_OF(CHAR , INTEGER))"
```

Meaning: The usual union of two sets.

INTER

Syntax: <set expression> INTER <set expression>
Type: SET_OF m1 X SET_OF m2 --> SET_OF m3
where m3 = m1 if m1 is the same as m2; generates an error if m1 and m2 refer to disjoint types, and m3 is the intersection of m1 and m2 otherwise.
Example:

```
If s1 is of type "SET_OF ONE_OF(INTEGER, BOOLEAN)"
and s2 of type "ONE_OF( SET_OF INTEGER, SET_OF CHAR)",
then s1 INTER s2 is of type "SET_OF INTEGER"
```

Meaning: usual set intersection.

DIFF

Syntax: set_1 DIFF set_2
Type: SET_OF m1 X SET_OF m2 --> SET_OF m1
Generates an error if m1 and m2 cannot have any common value.
Example:

If set_1 is of type "SET_OF ONE_OF(INTEGER, BOOLEAN)" and set_2 is of type "ONE_OF(SET_OF INTEGER, SET_OF CHAR)"), then the result will have the type of the first argument.

Meaning: $\{x \mid x \in \text{set\_1 AND } x \notin \text{set\_2}\}$

11.7.3. Predicates

INSET

Syntax: <expression> INSET <set_expression>
Type: m1 X SET_OF m2 --> BOOLEAN
Generates an error if m1 and m2 are totally disjoint types (as with DIFF).
Meaning: The expression is TRUE if the first operand is in the set represented by the second.

SUBSET

Syntax: set_1 SUBSET set_2
Type: SET_OF m1 X SET_OF m2 --> BOOLEAN
m1 and m2 are subject to the same constraints as with INSET.
Meaning: The expression is TRUE iff set_1 is a subset of set_2. Note that "s1 SUBSET s2" can be defined as :
FORALL x INSET s1: x INSET s2

11.7.4. Set Constructors

Explicit Constructor

Syntax: '{' expression_1,..., expression_n '}'
(Here, the curly brackets are actual symbols of the language.)
Type: If all the expressions have the same type, let us say t, then the result will be of type "SET_OF t"; otherwise, it will be "SET_OF ONE_OF(t1,..,tj)" where the united-type is the minimal union of the types of the expressions.
Value: The set S such that:
$\text{expression\_i} \in S \ (i = 1, 2,..,n)$

Implicit Constructor

Syntax: '{' <simple declaration> '|' <expression> '}'
Type: <expression> must be Boolean; result is of type "SET_OF t" where t is the type of the variable appearing in the declaration.
Value: "{t x | e(x)}" is the set of all x's of type t such that e(x) is TRUE.

11.8. Operations on Vectors

Extractor

Syntax: <vector expression> [<integer expression>]
Type: If the first operand is of type "VECTOR_OF t", the result will be of type t.
Value: v[i] is the ith component of the vector v.

LENGTH

Syntax: LENGTH(<vector expression>)
Type: VECTOR_OF mode --> INTEGER
Value: the number of components (dimensionality) of its operand.

Explicit Constructor

syntax: VECTOR(exp_1,...,exp_n)

Type: If exp_1,...,exp_n are all expressions of type t, then the result will be of type "VECTOR_OF t"; if the types are different, the result will be a vector of the minimal union of these types.

Value: V such that
LENGTH (V) = n and
V[i] = exp_i for i = 1, 2,..., n

Implicit Constructor

Syntax: VECTOR '(' FOR <symbol> FROM <expression>
TO <expression> ':' <expression> ')'

Type: If the last expression always has the type t, then the result will be a "VECTOR_OF t" (the expressions following FROM and TO must have the type INTEGER). Note that this is the only place in the language where a name is bound to an object that need not be previously declared; the symbol following FOR is a locally bound variable whose type is always INTEGER.

Value: V = VECTOR(FOR i FROM exp1 TO exp2 : f(i))
is the vector such that:
LENGTH(V) = exp2 - exp1 + 1 and
V[i] = f(exp1 + i - 1) for i = 1, 2,...,(exp2 - exp1 + 1)

11.9. Operations on Structures

Extractor

A reference to a particular field of a structure-expression (i.e., an expression whose type is a structured type) can be made by appending a '.' followed by the name of the particular field after the structure-expression itself. The syntax is thus:
<expression> '.' <symbol>

Type: The type is that associated with the field name in the structure declaration.

Meaning: The structure expression being a tuple, the extractor refers to the value of the nth element of this tuple, if the field name appeared in the nth position in the structure declaration.

Explicit Constructor

Syntax: '<' exp_1,...,exp_n '>'

Type: t1 X .... X tn
where ti is the type of the ith expression in the list.

Value: the tuple whose ith element is exp_i, for i from 1 to n.

Implicit Constructor

Syntax: '<' FOR <symbol> FROM <expression> TO <expression>
':' <expression> '>'

Type: t X .... X t
i.e., the n-fold Cartesian product of the type of the third

expression, where n is the difference between the second and first expressions.
Meaning: The tuple < exp_1,...,exp_n > where exp_i is the value of the third expression evaluated when the indicial variable (i.e., the <symbol>) has the value of the expression following FROM, augmented of i-1.

11.10. Quantified Expressions

Although quantified expressions are constrained to yield a Boolean value, they have been placed in this separate section because of their particular syntax rules.

Universal Quantifier

```
Syntax:  FORALL {<qualification> | <declaration>}
                { ";" {<qualification> | <declaration>} }*
                ":" <expression>

with:  <qualification> ::= {<simple declaration> | <symbol>}
                                   {INSET | "|"} <expression>
```

Type: BOOLEAN
Meaning: as in predicate calculus
Remarks: The <expression> following the ":" will presumably depend on the variables declared after the quantifier. These variables are bound locally, only in the last expression (see Scope Rules, Section 12). The optional qualification may be used to further qualify the variable, without complicating the main quantified expression.
The general way of expressing something such as "for all x's such that P(x), Q(x) is true" is:

```
FORALL x | P(x) : Q(x)
```

whereas "for all x's in the set S, Q(x) is true" would be written as:

```
FORALL x INSET S : Q(x)
```

(Note that, in the two examples above, we used the deferred binding facility, presuming that x appeared in the DECLARATIONS paragraph.)
Several variables may be quantified simultaneously, e.g.,

```
FORALL x | P(x) ; y | Q(y) ; z : R(x, y, z)
```

In such a case, each quantified variable may appear in its own qualification and in the main (quantified) expression, but not in the qualification of another variable.
The above expression has the following meaning:

```
FORALL x : FORALL y : FORALL z : P(x) AND Q(y) => R(x, y, z)
```

Existential Quantifier
Syntax:

```
EXISTS {<qualification> | <declaration>}
         { ";" {<qualification> | <declaration>} }*
         : <expression>
```

Type: BOOLEAN
Meaning: as in predicate calculus
Remarks: same as above.

11.11. Conditional Expressions

Syntax: IF <expression> THEN <expression> ELSE <expression>
Type: Consider the expression

IF P THEN e1 ELSE e2

If e1 and e2 have the same type, that type will be the type of the whole expression; if one of e1 and e2 (or both) is UNDEFINED or ?, then the type will match anything. If the types of e1 and e2 do not match, then the type of the expression will be the union of the type of e1 and the type of e2.
Meaning: x = IF P THEN e1 ELSE e2 is an abbreviation for (P AND (x = e1)) OR ((NOT P) AND (x = e2)
Remark: The ELSE part must always be present!

11.12. TYPECASE Expressions

This kind of expression is probably the most complex construct of the language. It is intended to be used in cases where the type of an object used in the system being specified cannot be known before execution time. Its purpose is to allow a temporary alteration of the type of an object, from a united type to one of the component types. This is particularly desirable because it permits the application to the object of some operations that would not be defined on the united type without compromising the safety of type-checking.

Syntax:

```
TYPECASE <symbol>  OF
type_1:  exp_1;
type_2:  exp_2;
      .
      .
      .
type_n:  exp_n;
END
```

Type: The type of the whole expression will be the minimal union of the types of exp_1,..., exp_n, in the sense that, if exp_1,.. exp_n all have the same type t, then t will be the type of the expression, otherwise, the type will be ONE_OF(t1, t2,...,tk) where t1,..,tk are all different, and are types of exp_1,...,exp_n (in no particular order).

Meaning: The value of the expression is more easily understood by using the meta-function "type_of" the value of which would be the type of its single argument. Then, for instance

```
a = TYPECASE w of
type_1:  exp_1;
type_2:  exp_2;
type_3:  exp_3;
END;
```

would mean

```
((type_of(w) = type_1) AND a = exp_1)
OR ((type_of(w) ~= type_1
      AND type_of(w) = type_2)
    AND a = exp_2)
OR ((type_of(w) ~= type_1
       AND type_of(w) ~= type_2
       AND type_of(w) = type_3)
    AND a = exp_3)
```

Remarks: There are several restrictions imposed on the use of this construct. We will call

* case variable the variable following the keyword TYPECASE;
* case labels the types type_1,..., type_n;
* case expressions exp_1,..., exp_n;

* The type of the case variable must be a united type equal to the type that would be obtained by taking the union of all the case labels.

Example: If w is of type

```
ONE_OF(INTEGER, BOOLEAN, capability)
```

then

```
TYPECASE w of
INTEGER:  w;
BOOLEAN:  IF w = TRUE THEN 1 ELSE 0;
capability:  get_uid(w);
END;
```

is a valid expression;

```
TYPECASE w of
ONE_OF(INTEGER, BOOLEAN): TRUE;
capability:  FALSE;
END;
```

is also valid.

(As an illustration, let us mention that the first expression would have the type ONE_OF(INTEGER, "type_of get_uid"), whereas the second one would have the type BOOLEAN.)

On the contrary,

```
  TYPECASE w OF
  INTEGER: 1;
  BOOLEAN : 0;
  END;
```

is incorrect because one of the types is missing.

* Since the expression itself has a type, regular type checking is performed.

* When the type of each case expression is evaluated, if the case variable appears in the expression, it is taken to have the type of the corresponding case label.

11.13. LET Expressions

It is sometimes necessary to bind some variables locally inside a particular expressions. Yet their use is generally very limited. When it seems to be really necessary, the user can declare a list of variables immediately before using them; their scope is restricted to one single expression, and the syntax for this construct is:

```
LET <qualification> { ";" <qualification> }*
        IN <expression>
```

The value and the type of this expression are those of the expression following the keyword IN. The closest approximation to the semantics of:

```
y = LET x | P(x) IN Q(x)
```

is something like

$\exists$ x such that P(x) AND (y = Q(x))

but this is only an example of the use of this construct.

11.14. SOME Expressions

The SOME construct can be viewed as a kind of set extractor. The syntax is:

```
SOME <simple declaration> {INSET | "|"} <expression>
```

The meaning of an expression such as

```
        SOME x INSET S

    is: LET x INSET S IN x
```

and similarly if "|" is used.

The following example illustrates the use of this construct:

```
        EFFECTS_OF wake_up( SOME process p
                                 INSET waiting_processes(sem1))
```

which might be an assertion inside the specification for a synchronization primitive.

11.15. The NEW Primitive

NEW is a special construct that has the same syntax as a function of one argument:

```
        NEW '(' <symbol> ')'
```

where the argument should be the name of a designator type. The type of this expression is precisely this identifier, and the meaning is that NEW(t) represents an object of type t that has never been used before.

An error will occur if the argument is not a DESIGNATOR (even an external DESIGNATOR will not work).

11.16. Miscellaneous Operators

In addition to the various operators and constructs presented above, the language contains several functional primitives.

MAX, MIN
Syntax: {MAX | MIN} '(' <expression> ')'
Type: SET_OF number --> number
(where number is either INTEGER or REAL)
Value: the largest (respectively smallest) element in the set described in the expression.

SUM
Syntax: SUM '(' <expression> ')'
Type: SET_OF number --> number
or: VECTOR_OF number --> number
Value: the arithmetic sum of all the elements in the set (respectively vector).

INTPART, FRACTPART
Syntax: {INTPART | FRACTPART} '(' <expression> ')'
Type: REAL --> INTEGER (resp. REAL)
Value: These two constructs can be viewed as predefined macros:

```
INTEGER INTPART( REAL x ) IS SOME INTEGER y | y <= x  AND y+1
> x;
REAL FRACTPART( REAL x ) IS x - INTPART( x );
```

They define, respectively, the integer and fractional part of their argument (surprised?).

******

This concludes our syntactic description of the language for module specifications. We will now concern ourselves with some additional precisions concerning the use of certain objects.

12. PRECEDENCE

The large number of operators in the assertion language requires a precise definition of the precedence ordering, that is, if op1 and op2 are two operators (let us say binary), we say that op1 has a higher precedence than op2 if

    e1 op1 e2 op2 e3

has the same meaning as

    (e1 op1 e2) op2 e3

and

    e1 op2 e2 op1 e3

has the same meaning as

    e1 op2 (e2 op1 e3)

(Note that any expression can always be enclosed within parentheses.)

A general rule is that an operator returning a Boolean valued expression has always a lower precedence than an operator that returns another type. This rule is of course insufficient to compare operators returning the same type of expression.

The ordering is partial, since the operators cannot be freely combined in the text (e.g., INTER can hardly be compared with +).

The operators are listed in decreasing order of precedence. Constructs that do not lead to ambiguities (reference to functions, LENGTH, CARDINALITY, NEW, vector subscripting, TYPECASE, FORALL, EXISTS, vector and set constructors, LET, and SOME) are not listed, but it should be noted that, when used as operands with any of the operators listed below, the constructs using IF, LET, SOME, FORALL, EXISTS, and TYPECASE should be enclosed within parentheses.

<u>INTEGER</u>	<u>BOOLEAN</u>	<u>sets</u>
unary -		
*, /, MOD		INTER
+, -		UNION, DIFF
	=, ~=, <, <=, >=, >, INSET, SUBSET	
	NOT	
	AND	
	OR	
	=>	

Of interest are the places of NOT, INSET and => : thus

(NOT x) INSET S

is meaningful only if x has the type BOOLEAN and s is a set of BOOLEAN (for instance {TRUE})

NOT x INSET s => P(x)

means (x INSET s) OR P(x) .

13. SCOPE RULES

A name designates an object by two mappings: one from the name to a type and the other from the name to a "binding" (in the sense described in Section 4).

The basic rule is that a name cannot be used in an expression if the value of either of the two mappings is undefined.

The dual possibility of associating a name with a type through a declaration has been described at length in Section 4 and is not repeated here.

13.1. <u>Establishing the Binding</u>

The binding of an object is established by declaring the name one wishes to associate with the object in a predetermined place in the text, which depends on the category of the object itself (by "declaring", we mean that the name can appear in a declaration, in a header, or in the place of a declaration in the case of deferred binding).

Here is a table that indicates the place where a binding is established.

<u>Object</u>	<u>Binding</u>
Global definition	Definition in the DEFINITIONS paragraph
Local definition	Definition in the DEFINITIONS section of a function
Formal argument	When appearing in a function or macro header
Result argument	"
Quantified variable	When appearing after FORALL and EXISTS
Locally bound variable	When appearing between '{' and '|', or after FOR, LET or SOME

13.2. <u>Scope of a Binding</u>

We define the scope of a name as that part of a module specification where the name may be used in an expression, independently of the constraints imposed by type compatibility.

Generally, a binding is valid from the place where it was established up to some precise place, given below (although in some cases we give the scope itself rather than the place where the binding is destroyed).

<u>Object</u>	<u>Binding</u>
Global definitions	The whole module
Formal and result arguments	Valid only in the specification of the corresponding function or in the expression defining the macro
Locally bound variable	The expression(s) following the '|', INSET or ':' after FORALL, EXISTS, LET, SOME, and in set and vector constructors.

An important additional comment is that, although it is permitted to alter a valid binding temporarily (e.g., by using the name of a formal argument as a locally bound variable), this will mask the original binding for the scope of the new one. It seems that this practice should be avoided, inasmuch as it can bring only confusion in the specification.

14. COERCION RULES

As has been said in Section 5, a united type is very close to being a type by itself, and normally an expression of a united type should not match an expression of one of the components of the united type. However, such compatibility is permitted in some cases. The determination of these cases is based on what is known as the "coercion rules".

The first case of coercion is the TYPECASE expression, which should be the normal (and only) way to coerce "downwards", i.e., from a united type to a component type. Upward coercion (i.e., when an expression of a simple type is implicitly converted to a united type containing the original type as one of its components) may take place in the following cases:

* When a formal argument to a function is of a united type, the corresponding actual argument may be of a component type.

* In an expression of the form a = b, where a is a _quoted_ V-function reference, the result argument of the V-function being of a united type, and b is an expression of a component type.

The actual rule is the same as above for TYPECASE expressions and actual arguments to functions. It is more general in the other cases, where an attempt to match a united type against one of its components is tolerated, albeit with a warning message which will hopefully remind the user that he/she is walking on dangerous ground. A warning message will also be issued when, in the constructs IF, SET, and VECTOR, the types of the constituent expressions do not match. However, this will not be the case for the binary operators on sets, UNION, INTER, DIFF, INSET, and SUBSET.

15. COMMENTS

Although it has been barely mentioned so far, the language has a comment feature: A comment consists of the sign '$' followed by a sequence of characters without any space, parentheses, or square bracket; or by a string of characters enclosed within two double-quote signs (") (the string itself containing no double-quote); or by any sequence of characters enclosed within two matching parentheses or square brackets (in the case of parentheses, the sequence must not contain any right square bracket that is not preceded in the sequence by a left one). More briefly, a comment can be described as a '$' followed by any arbitrary LISP S-expression--see (Tel 75).

A comment may appear at any place where a space is legal (i.e., anywhere except in the middle of a syntactic unit).

16. MAPPING FUNCTION EXPRESSIONS

The specifications for mapping function expressions follow a slightly different syntax, although the "expression" part of the language is exactly the same.

16.1. Module Names

A list of mapping functions specifications should begin with

```
MAP module_1 TO module_2,..., module_k;
```

where module_1 is the module containing the functions and parameters to be expressed as functions of elements of module_2,...,module_k; module_1 is referred to as the "target" module.

16.2. TYPES

The TYPES paragraph is similar to the one described for module specifications, and it abides by the same rules.

16.3. DECLARATIONS

The same conventions for deferred binding can optionally be applied and the DECLARATIONS paragraph provides the same facility towards this end.

16.4. DEFINITIONS

The same macro capabilities are provided in the DEFINITIONS paragraph.

16.5. EXTERNALREFS

An EXTERNALREFS paragraph similar to the one optionally used in a module specification is mandatory for mapping functions; it must contain declarations for all the primitive objects (i.e., designator and scalar types, parameters, and V-functions that are not derived) appearing in the mapping function.

16.6. MAPPINGS

The MAPPINGS paragraph begins with MAPPINGS. It contains a list of pairs of the form

```
<object> :  <expression>;
or
<symbol> : <type specification>;
```

where <expression> is any expression satisfying the rules described earlier, and <object> is one of the following:

* The name of a parameter of the target module

* The name of a scalar type, in which case <expression> must be a set expression consisting of a list of as many expressions as there are constants in the scalar type definition;

* A construct of the form

  <symbol> (<argument list>)

where <symbol> is the name of a visible V-function or a parameter (with arguments) of the target module, and <argument list> is a list of formal arguments in the usual sense.

The same scope rules apply, except that there is no result argument.

The type of each expression, derived from the types of its various components declared either in the declarations or in the source modules, should be the same as that of the associated parameter or V-function in the target module, or a set expression in the case of a scalar type name.

In the second form, the symbol should be the name of a designator type of the target module, and the type specification should be legal in at least one of the lower modules.

17. THE TENEX-INTERLISP IMPLEMENTATION

The properties of specifications can be checked automatically by a "specification handler". Such a program has been written to run under the TENEX operating system on a PDP-10 machine; some of the conventions used (e.g., for strings) are a consequence of the conventions used in the implementation language (namely INTERLISP).

17.1. Strings

A string is the equivalent of an INTERLISP string, i.e., an arbitrary sequence of printable ASCII characters enclosed within two " (double-quote), where the characters " and % have to be represented as %" and %%.

17.2. Identifiers

An identifier (generally represented as <symbol> in this manual) is any sequence of up to 126 printable ASCII characters which is not a number, which does not contain any of the characters (,), [,], {, }, |, ,, ;, :, =, ~, -, +, <, >, *, /, ', ¬, ?, $, space; and which does not start with either " or #. Note that 1%&@! is a perfectly legal identifier.

17.3. File Inclusion

If the character # appears after any of the characters listed in the previous paragraph, the next expression (in the INTERLISP sense) will be interpreted as a file name, and the program will take its input from that file until an end of file is reached, at which point it will come back to the original file. This may be repeated, i.e., the included file may itself include a third one, etc., although a file may not include itself, directly or indirectly.

18. CONCLUDING REMARKS

The reader may have noticed (by the very absence of such expressions) that nowhere have we spoken of "the value of a variable" or "the contents of a variable". Such phrases have been (carefully) avoided to stress one of the particularities of any specification language, i.e. the property of being nonprocedural; a name refers to a mathematical object, and never to an address or any other component of a computer. Indeed the language does not provide the concept of a pointer, and this could be introduced only as a designator.

SPECIAL is often modified to deal with previously unsuspected problems. Part of the work to be done in order to design a good specification language is to understand fully the concepts involved and the specification methodology on which the language is based. A very desirable goal would also be the complete axiomatization of the language constructs, similar to the work done on Pascal by Hoare and Wirth (Hoa 73). We hope to come up with such a formalization in the (more or less) near future.

ACKNOWLEDGEMENTS

The authors are grateful to the various members of the Computer Science Group at SRI for several suggestions. Of particular importance were the contributions of Dr. Robert S. Boyer. An important part of the earlier version of SPECIAL was the work of Bernard Mont-Reynaud.

The work described here has been supported by the US Air Force under Honeywell Purchase Order #9984718, by the Naval Electronics Laboratory Center under Contract N00123-76-C-0195, and by the National Science Foundation under Grant DCR74-18661.

- REFERENCES -

(Hoa 73) Hoare,C.A.R., and Wirth,N.; "An Axiomatic Definition of the Programming Language Pascal," Acta Informatica, vol.2, (1973).

(Neu 74) Neumann,P.G. et al.; "On the Design of a Provably Secure Operating System," Proc. Workshop on Protection in Operating Systems, IRIA, Rocquencourt, (August 1974).

(Neu 75) Neumann,P.G., et al.; "A Provably Secure Operating System," Final Report, Project 2581, Stanford Research Institute, Menlo Park, California, (June 1975).

(Par 72 a) Parnas,D.L.; "A Technique for Software Module Specification with Examples," Comm. ACM, vol.15 (May 1972).

(Par 72 b) Parnas,D.L.; "On the Criteria To Be Used in Decomposing Systems into Modules," Comm. ACM, vol.15 (December 1972).

(Rob 75 a) Robinson,L. et al.; "On Attaining Reliable Software for a Secure Operating System," Proc. 1975 International Conference on Reliable Software, Los Angeles, (April 1975).

(Rob 75 b) Robinson,L., and Levitt,K.N.; "Proof Techniques for Hierarchically Structured Programs," Stanford Research Institute (to be published).

(Tei 75) Teitelman,W.; "INTERLISP Reference Manual," XEROX Palo Alto Research Center, (December 1975).

APPENDIX A : GRAMMAR

Legend: The grammar for the specification language is given in the so-called extended BNF, where <...> means that the enclosed symbol is a nonterminal of the grammar; [...] means that the enclosed construct is optional; {...}* means that the enclosed construct can occur 0 or more times; {...}+ means 1 or more times, and {... ! ... ! ...} means an alternation, i.e., any of the constructs listed can be used.

For convenience, all the nonalphabetic terminals have been enclosed within simple quotes (').

<symbol> stands for any identifier, <number> for any integer or real number.

```
ROOT                       ::= MODULE <symbol> [<types>] [<declarations>]
                                   [<parameters>] [<definitions>]
                                   [<externalrefs>] [<functions>]
                                   END_MODULE
                           ::= MAP <symbol> TO <symbol> { ',' <symbol> }* ';'
                                   [<types>] [<declarations>] [<parameters>]
                                   [<definitions>] [<externalrefs>]
                                   [<mappings>] END_MAP

<types>                    ::= TYPES { <typedeclaration> ';' }+

<typedeclaration>          ::= <symbol> { ',' <symbol> }* ':'
                                   { DESIGNATOR | <typespecification>
                                     | <setexpression> }

<typespecification>        ::= <symbol>
                           ::= INTEGER
                           ::= BOOLEAN
                           ::= REAL
                           ::= CHAR
                           ::= STRUCT '(' <declarationlist> ')'
                           ::= ONE_OF '(' <typespecification>
                                   { ',' <typespecification> }+ ')'
                           ::= { SET_OF | VECTOR_OF } <typespecification>

<simple declaration>       ::= <typespecification> <symbol>

<declaration>              ::= <simple declaration> { ',' <symbol> }*
                           ::= <symbol>

<declarations>             ::= DECLARATIONS { <declaration> ';' }+

<parameters>               ::= PARAMETERS { <parameterdeclaration> ';' }+

<parameterdeclaration>     ::= <typespecification> <symbol> [<formalargs>]
                                   { ',' <symbol> [<formalargs>] }

<formalargs>               ::= '(' [<declarationlist>] ')'
                                   [ '[' <declarationlist> ']' ]

<declarationlist>          ::= <declaration> { ';' <declaration> }*

<definitions>              ::= DEFINITIONS { <definition> ';' }+
<definition>            ::= <typespecification> <symbol> [<formalargs>]
                              IS <expression>

<externalrefs>          ::= EXTERNALREFS { externalgroup }+

<externalgroup>         ::= FROM <symbol> ':' { <externalref> ';' }+

<externalref>           ::= <parameterdeclaration>
                        ::= <symbol> { ',' <symbol> }* ':' DESIGNATOR
                        ::= <symbol> ':' <setexpression>
                        ::= { VFUN | OVFUN } <symbol> <formalargs>
                                '->' <declaration>
                        ::= OFUN <symbol> <formalargs>

<functions>             ::= FUNCTIONS { <functionspec> }+

<functionspec>          ::= VFUN <symbol> <formalargs>
                              '->' <declaration> ';'
                              [<definitions>]
                              [{ HIDDEN ';' | <exceptions> }]
                              { INITIALLY | DERIVATION } <expression> ';'
                        ::= OVFUN <symbol> <formalargs>
                              '->' <declaration> ';'
                              [<definitions>] [<exceptions>] { <delay> }*
                              [<effects>]
                        ::= OFUN <symbol> <formalargs> ';' [<definitions>]
                                [<exceptions>] { <delay> }* [<effects>]


<delay>                 ::= DELAY UNTIL <expression> ';'

<exceptions>            ::= EXCEPTIONS { <expression> ';'
                                | EXCEPTIONS_OF <call> ';' }+

<effects>               ::= EFFECTS { <expression> ';' }+

<mappings>              ::= MAPPINGS { <mapping> ';' }+

<mapping>               ::= <symbol> [<formalargs>] ':' <expression>
                        ::= <symbol> ':' <typespecification>

<expression>            ::= IF <expression> THEN <expression>
                              ELSE <expression>
                        ::= LET <qualification> { ';' <qualification> }*
                              IN <expression>
                        ::= SOME <qualification>
                        ::= { FORALL | EXISTS } <qualif\declarationlist>
                              ':' <expression>
```

```
                         ::= TYPECASE <symbol> OF { <case> ";" }+ END
                         ::= <expression> <binaryoperator> <expression>
                         ::= { NOT | "~" } <expression>
                         ::= "-" <expression>
                         ::= "(" <expression> ")"
                         ::= <symbol>
                         ::= <number>
                         ::= <character constant>
                         ::= <string constant>
                         ::= TRUE | FALSE | UNDEFINED | ?
                         ::= <expression> "[" <expression> "]"
                         ::= <expression> "." <symbol>
                         ::= { CARDINALITY | LENGTH | MAX | MIN | SUM
                                 | INTPART | FRACTPART }
                                 "(" <expression> ")"
                         ::= NEW "(" <symbol> ")"
                         ::= [EFFECTS_OF] <call>
                         ::= <structureconstructor>
                         ::= <vectorconstructor>
                         ::= <setexpression>

<qualif\declarationlist>::= {<qualification> | <declaration>}
                                { ";" <qualification> | <declaration> }*

<qualification>          ::= [<typespecification>] <symbol> { "|" | INSET }
                                 <expression>

<case>                   ::= <typespecification> ":" <expression>

<binaryoperator>         ::= "*" | "/" | INTER | "+" | "-" | UNION | DIFF |
                                 "=" | "~=" | ">" | ">=" | "<" | "<=" |
                                 INSET | AND | OR | SUBSET | MOD | "=>"

<call>                   ::= ["'"] <symbol> "(" [ <expression>
                                 { "," <expression> }*] ")"

<structureconstructor>   ::= "<" [ <expression> { "," <expression> }* ] ">"
                         ::= "<" <range> ":" <expression> ">"

<vectorconstructor>      ::= VECTOR "(" [ <expression>
                                 { "," <expression> }* ] ")"
                         ::= VECTOR "(" <range> ":" <expression> ")"

<range>                  ::= FOR <symbol> FROM <expression> TO <expression>

<setexpression>          ::= "{" [ <expression> { "," <expression> }* ] "}"
                         ::= "{" [<typespecification>] <symbol> "|"
                                 <expression> "}"
```

APPENDIX B : RESERVED WORDS

The following list contains all the reserved words of the language, i.e., the identifiers that may NOT be used to designate arbitrary objects. The identifiers T and NIL, although not reserved, are not allowed in the current implementations.

AND
BOOLEAN
CARDINALITY
CHAR
DECLARATIONS
DEFINITIONS
DELAY
DERIVATION
DESIGNATOR
DIFF
EFFECTS
EFFECTS_OF
ELSE
END
END_MAP
END_MODULE
EXCEPTIONS
EXCEPTIONS_OF
EXISTS
EXTERNALREFS
FALSE
FOR
FORALL
FRACTPART
FROM
FUNCTIONS
HIDDEN
IF
IN
INITIALLY
INSET
INTEGER
INTER
INTPART
IS
LENGTH
LET
MAP
MAPPINGS
MAX
MIN
MOD
MODULE
NEW
NOT
OF
OFUN
ONE_OF
OR
OVFUN
PARAMETERS
REAL
SET_OF
SOME
STRUCT
SUBSET
SUM
THEN
TO
TRUE
TYPECASE
TYPES
UNDEFINED
UNION
UNTIL
VECTOR
VECTOR_OF
VFUN

Appendix C*

COMMENTS ON "SECURITY KERNEL SPECIFICATION FOR SECURE COMMUNICATIONS PROCESSOR"

(CDRL A018)

Contract F19628-74-C-0193,
Documents Review 1 February 1977
Department of the Air Force
Headquarters Electronics Systems Division (AFSC)
Hanscom Air Force Base, Massachusetts 01731

*Appendix C contains only those Air Force comments which remain pertinent. Many comments have been addressed in the current specification.

The top level specification shows significant progress on the part of Honeywell. In general, the functions provided in specification appear adequate. However, there are several potential difficulties with the current specification. These difficulties relate to the specification detail and style and the specific design decisions.

The specifications in their current state are more detailed than necessary for a top level specification. The top level specification need specify only what the kernel does. The current specification indicates how the kernel is implemented. Specifying too much detail has the drawback of complicating the proof of correspondence between the kernel and model elements. The extra detail makes the specifications longer which, in turn, make the proofs longer and more complicated. Also more detailed design decisions are more subject to change. Changes to the kernel design after the proofs could be costly because significant effort may be required to reprove the specification.

The detail of the current specification is not consistent. The report provides great detail about some aspects of the kernel but lacks detail regarding other aspects for no apparent reason. For example, the report details the effect of encountering a missing page (in spite of the claim that paging is invisible) but leaves out other details.

The memory manager interface does not appear adequate to permit implementation of typical memory management policies. The information available to the memory manager may not be sufficient to implement page management strategies such as least recently used or first-in, first-out. There may be additional data which would be helpful and not affect security. The current scheme appears to restrict the memory manager to random replacement policy.* Since the memory manager is a system high process, it seems that the memory manager could have access to any data maintained by the kernel. However, the memory manager's ability to modify kernel maintained data has to be controlled. Typical page replacement strategies employ sophisticated data bases (e.g. lists, queues, circular lists, etc.) and must monitor information from one invocation to another (e.g. pointers to where the search for candidates for removal ended on the last invocation, are often maintained, and used bits are turned off so that page use between successive invocations of the page replacement activity can be considered). The memory manager cannot reasonably obtain such information with the current interface. In fact, a memory manager interface that is both general and efficient many not be implementable.

*Since the kernel resets the page used indicators at each invocation of get_memory_data, the memory manager may select pages not used since previous invocation for swapping (see Section 4.6 for details).

The report claims that the memory manager is outside the kernel and can be implemented as an untrusted process. This claim is not readily apparent since the report does not provide a proof that the security provided by the kernel is independent of the memory manager and because the correct behavior of the kernel (the kernel's ability to fulfill its specification) depends on the memory manager. The kernel function get_memory_data provides the memory manager with system high information (the security level of the used and modified bits is the level of the process that caused the bits to be set). The add_pages_to_free_list is an interpretive write of kernel data. The level of the data provided to the kernel is system high. It must be shown that the data provided to the kernel is not visible to a lower level process. Because the correct operation of the kernel depends on the memory manager, the memory manager perhaps requires different treatment than other untrusted processes for policy reasons. Programs outside the kernel (not including programs upon which the kernel depends) should not be able to affect the operation of the memory manager. (For example, the memory manager could operate in a more privileged ring than that of the operating system, and the "core" security kernel could protect software in this ring just as it protects the software of its own ring.)

The previous paragraphs appear to conclude that the approach taken for the memory manager does not remove the memory manager from the kernel. Instead, the approach may simplify verification of the kernel (assuming the task of demonstrating that information returned to the kernel by the memory manager cannot be passed downward is simpler than proving correctness of the memory manager).

The rationale for not allowing process creation by untrusted processes is not apparent. The kernel must provide a capability to create and delete processes. The additional effort to permit untrusted processes to use these functions appears minimal and may result in other simplifications of the kernel such as the answering service.

There is a requirement that the top level specification for the SFEP provide multilevel network support. Functions in support of a multilevel network can be written in a general manner without a specific network in mind.

More specific comments are as follows:

Get_memory_data, add_pages_to_free_list - These functions provide for memory management. However, the exception should include a check for the caller being at system high.

Get_device_data - Since devices are not shared or pooled in the SFEP as are memory pages, there may be no use for this function or the device manager.

Async_device_read and async_device_write - Since these are hardware functions, their specification must accurately reflect the hardware. Are the first two exceptions that the device must be active and that the segment number hasn't changed, actually checked by the SPM?* Because these functions are performed asynchronously, the exceptions may have to be considered different from those of other functions.

Device_wake-up - The effects of this function are very much like the effects that may occur in async_device_read and async_device_write when asynchronous exception conditions exist.

Send_message, receive_message - Cannot the message itself be an argument to the function? That would eliminate most of the exceptions and some of the effects. True, the message data will probably not move from its original location in the user's segment, but most of the exceptions listed are those performed by argument validation anyway, the details of which are never specified.

The host-SFEP communication scheme lacks a method for communication between system processes that may not have a device as a link. Can one at this time rule out the need for a more general process-to-process communication link? The MITRE specification employs a process-to-process communication, with more generality and greater specification simplicity. When one considers the possibility of an untrusted single level network process in the SFEP that may have to communicate with more than one Multics process, the generality may be necessary.

* The exceptions shown are checked by the SPM hardware.